

XXL-JOB源码原理探究

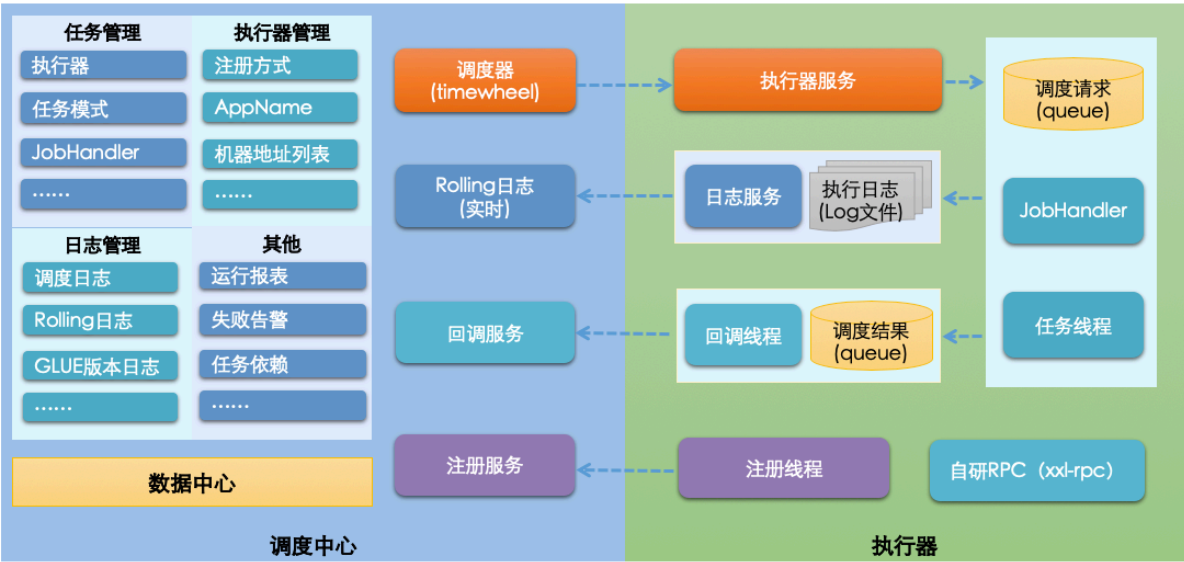
此项目是开源项目，是一个分布式调度的平台

支持分片执行任务，支持良好的扩展。项目源码地址戳[这里](#)，我们来分析下其中的调度原理和源码流程

在实际的业务开发中，定时任务是一个无法绕过的组件，我们在很多个业务场景中都需要用到。组件可供选择的也有很多，比如spring-schedule和强大的quartz, 其实原理大同小异，这篇文章会先研究下xxl-job的原理，后面会给出和其他定时任务组件的对比情况，会涉及到不同组件设计的理念和能解决的问题，从设计方面和架构方面来讨论什么是一个好的调度任务组件/框架。

模块和划分

xxl-job下面简称该组件，总的来说分为，任务管理，任务调度，任务执行。过程中伴随着各种日志数据的产生，并且涉及到RPC的注册和invoker调用，给出官方的2.1.0版本的架构图



笔者认为任务管理可以单独的作为一个模块来划分出去。除此之外目前的log方式是本地log的记录，推荐开放log组件扩展，可以让业务方式监控更加有效和提供高效的数据收集。

具体的调度和执行流程

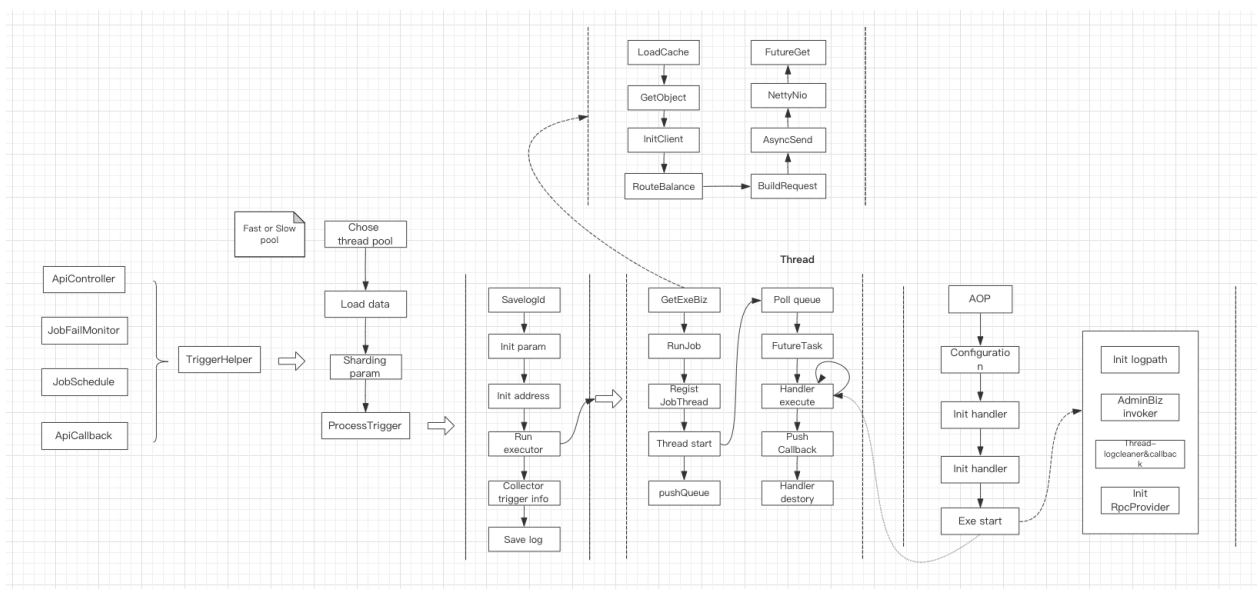
我们来看下代码到底是怎么做的，其实学习任何一个设计或者组件都需要先从更高的视角来识别。

资源准备

- Job配置 (spring->xml spring-boot->beanConfiguration)
- Job的method或者class注册
- 生产资料的准备，包括初始化调度线程资源和log配置
- Job线程execute
- 初始化RPC并注册到server上

called

- 调用的方式-> api/failedmonitorThread/schedule/apiCallback
- choseThreadPool (线程池之间的隔离，目前只有快慢隔离)
- 根据不同的路由策略获取执行方的具体地址
- 封装client参数rpc调用
- 具体执行机器被call, 执行具体的逻辑，return result



源码

First, 先看下Job的准备部分的代码

```
@Bean // spring初始化该bean
public XxlJobSpringExecutor xxlJobExecutor() {
    XxlJobSpringExecutor xxlJobSpringExecutor = new XxlJobSpringExecutor(); //
    new对象
    xxlJobSpringExecutor.setXXX();
    return xxlJobSpringExecutor;
}
```

```

public class XxlJobSpringExecutor extends XxlJobExecutor implements
ApplicationContextAware, InitializingBean, DisposableBean { // 该类实现了aware和
InitializingBean,在spring加载后就会实例化

    @Override
    public void afterPropertiesSet() throws Exception { //在构造方法之前会被调用
        // init JobHandler Repository
        initJobHandlerRepository(applicationContext); // 注册aop的
        annotation@JobHandler

        // init JobHandler Repository (for method)
        initJobHandlerMethodRepository(applicationContext); // 注册aop的
        annotation@XxlJob

        // refresh GlueFactory
        GlueFactory.refreshInstance(1);

        // super start
        super.start(); // 重要的初始化步骤, 调用super的start方法
    }
    // xxxFunc()
}

```

进入super的start方法

```

public void start() throws Exception {
    // 1. init logpath
    XxlJobFileAppender.initLogPath(logPath);

    // 2. init invoker, admin-client
    initAdminBizList(adminAddresses, accessToken);

    // 3. init JobLogFileCleanThread
    JobLogFileCleanThread.getInstance().start(logRetentionDays);

    // 4. init TriggerCallbackThread. 无论threadJob执行任务失败还是成功都会push
    callBack queue数据
    TriggerCallbackThread.getInstance().start();

    // 5. init executor-server, 重要的initRpcServer
    port = port>0?port: NetUtil.findAvailablePort(9999);
    ip = (ip!=null&&ip.trim().length()>0)?ip: IpUtil.getIp();
    initRpcProvider(ip, port, appName, accessToken);
}

```

进入initRpcProvider方法, 该方法就是把当前机器的Ip & port注册到数据中心, 在job执行的时候等待被调度到, 省略部分逻辑

```

private void initRpcProvider(String ip, int port, String appName, String
accessToken) throws Exception {
    // serviceRegistryParam, appName & address
    xxlRpcProviderFactory = new XxlRpcProviderFactory();
    xxlRpcProviderFactory.setXXX(); // set部分省略
    // add services, 把serviceKey -> ExecutorBiz的实例化对象放入map
    xxlRpcProviderFactory.addService(ExecutorBiz.class.getName(), null, new
ExecutorBizImpl());
    // start
    xxlRpcProviderFactory.start(); // 重要的server start
}

```

接下来进入xxlRpcProviderFactory.start()方法，马上就要到了终点。

```

private Class<? extends Server> server = NettyServer.class; // 这里看到server是
NettyServer的实例
//...省略部分

private Server serverInstance;
private Serializer serializerInstance;
private ServiceRegistry serviceRegistryInstance;
private String serviceAddress;
// ...省略部分

public void start() throws Exception {
    // ... 省略部分校验和set
    serverInstance = server.newInstance();
    serverInstance.setStartedCallback(new BaseCallback() { // serviceRegistry
started
        @Override
        public void run() throws Exception {
            // start registry
            if (serviceRegistry != null) {
                serviceRegistryInstance = serviceRegistry.newInstance();
                serviceRegistryInstance.start(serviceRegistryParam); // 把参数注册到
api/registry url上
                if (serviceData.size() > 0) {
                    serviceRegistryInstance.registry(serviceData.keySet(),
serviceAddress);
                }
            }
        }
    });
    serverInstance.setStopedCallback(new BaseCallback() { // serviceRegistry
stoped
        @Override
        public void run() {
            // stop registry

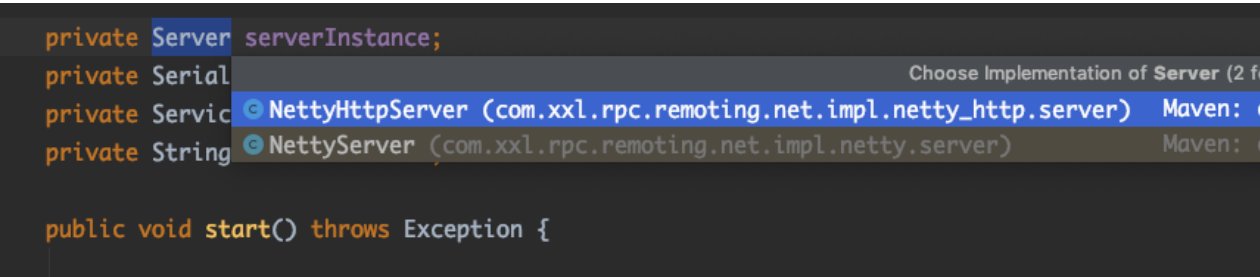
```

```

        if (serviceRegistryInstance != null) {
            if (serviceData.size() > 0) {
                serviceRegistryInstance.remove(serviceData.keySet(),
serviceAddress);
            }
            serviceRegistryInstance.stop();
            serviceRegistryInstance = null;
        }
    }
});
serverInstance.start(this); // NettyServer执行start
}

```

在这里serverInstance是Server抽象类的实例，我们看下图，Server的两个继承分别是NettyHttpServer和NettyServer，这两个类很重要，是执行Job调度的底层通信设施，从名字上我们也可以看出来，这里实现了Netty的封装，用Netty作为网络I/O的通信。



```

private Server serverInstance;
private Serial Choose Implementation of Server (2 f
private Service G NettyHttpServer (com.xml.rpc.remoting.net.impl.netty_http.server) Maven:
private String G NettyServer (com.xml.rpc.remoting.net.impl.netty.server) Maven:

public void start() throws Exception {

```

下面进入NettyServer执行start的逻辑，这里就进入了Netty的信道，关于Netty可以看我的另一个文章，在这里暂时不做深入，start启动后，job作为server的准备工作基本全部结束。从job的method aop开始，到serviceKey的注册，和RPCserver的初始化，整个核心流程基本清晰

```

@Override
public void start(final XmlRpcProviderFactory xmlRpcProviderFactory) throws
Exception {
    thread = new Thread(new Runnable() {
        @Override
        public void run() {
            // 处理event事件的线程池
            final ThreadPoolExecutor serverHandlerPool =
ThreadPoolUtil.makeServerThreadPool();
            try {
                // start server
                ServerBootstrap bootstrap = new ServerBootstrap();
                bootstrap.group(new NioEventLoopGroup(), new NioEventLoopGroup())
                    .channel(NioServerSocketChannel.class)
                    .childHandler(new ChannelInitializer<SocketChannel>() {
                        @Override
                        public void initChannel(SocketChannel channel) throws Exception {
                            channel.pipeline()
                                .addLast(new IdleStateHandler(0,0, Beat.BEAT_INTERVAL*3,
TimeUnit.SECONDS)) // beat 3N, close if idle

```

```

        .addLast(new NettyDecoder(XxlRpcRequest.class,
xxlRpcProviderFactory.getSerializerInstance()))
        .addLast(new NettyEncoder(XxlRpcResponse.class,
xxlRpcProviderFactory.getSerializerInstance()))
        // 这里绑定了执行request的handler, 下面我们会分析到
        .addLast(new NettyServerHandler(xxlRpcProviderFactory,
serverHandlerPool));
    }
})
    .childOption(ChannelOption.TCP_NODELAY, true)
    .childOption(ChannelOption.SO_KEEPALIVE, true);
// bind端口, 开始监听port的请求
ChannelFuture future =
bootstrap.bind(xxlRpcProviderFactory.getPort()).sync();
// 调用之前注册的xxlRpcProviderFactory.start(), 启动
onStarted();
// wait util stop
future.channel().closeFuture().sync();
} catch (Exception e) {
    // 部分省略
} finally {
    // stop
    try {
        serverHandlerPool.shutdown(); // shutdownNow
    } catch (Exception e) {
        logger.error(e.getMessage(), e);
    }
    try {
        workerGroup.shutdownGracefully();
        bossGroup.shutdownGracefully();
    } catch (Exception e) {
        logger.error(e.getMessage(), e);
    }
}
}
});
thread.setDaemon(true);
thread.start();
}

```

上面讲了job的注册逻辑和准备工作, 下面开始check下触发job和分布式环境下job调度的逻辑, 一共分为4种途径来触发job的执行,

API/JobScheduleHelper/AdminBizImpl.callBack()/JobFailMonitorHelper

```

// 1. api触发器
@RequestMapping("/trigger")

```

```

@ResponseBody
public ReturnT<String> triggerJob(int id, String executorParam) {
    if (executorParam == null) {
        executorParam = "";
    }
    JobTriggerPoolHelper.trigger(id, TriggerTypeEnum.MANUAL, -1, null,
executorParam);
    return ReturnT.SUCCESS;
}

// 2. JobScheduleHelper.start(), 定时任务的入口
else if (nowTime > jobInfo.getTriggerNextTime()) {
    // 1、trigger
    JobTriggerPoolHelper.trigger(jobInfo.getId(), TriggerTypeEnum.CRON, -1,
null, null);
    // 2、fresh next
    refreshNextValidTime(jobInfo, new Date());
    // next-trigger-time in 5s, pre-read again
    if (jobInfo.getTriggerStatus()==1 && nowTime + PRE_READ_MS >
jobInfo.getTriggerNextTime()) {
        // 1、make ring second
        int ringSecond = (int)((jobInfo.getTriggerNextTime()/1000)%60);
        // 2、push time ring
        pushTimeRing(ringSecond, jobInfo.getId());
        // 3、fresh next
        refreshNextValidTime(jobInfo, new Date(jobInfo.getTriggerNextTime()));
    }
}

// 3. AdminBizImpl.callBack(), callback入口
for (int i = 0; i < childJobIds.length; i++) {
    int childJobId = (childJobIds[i]!=null && childJobIds[i].trim().length()>0
&& isNumeric(childJobIds[i]))?Integer.valueOf(childJobIds[i]):-1;
    JobTriggerPoolHelper.trigger(childJobId, TriggerTypeEnum.PARENT, -1, null,
null);
    ReturnT<String> triggerChildResult = ReturnT.SUCCESS;
}

// 4. JobFailMonitorHelper.run(), 监控重试入口
if (log.getExecutorFailRetryCount() > 0) {
    JobTriggerPoolHelper.trigger(log.getJobId(), TriggerTypeEnum.RETRY,
(log.getExecutorFailRetryCount()-1), log.getExecutorShardingParam(),
log.getExecutorParam());

    XxlJobAdminConfig.getAdminConfig().getXxlJobLogDao().updateTriggerInfo(log);
}

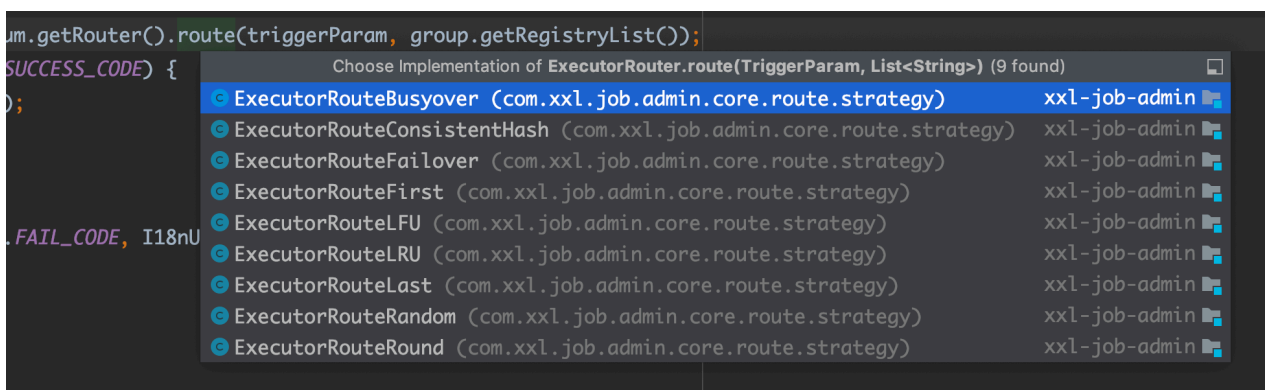
```

上面4种触发Job任务的途径统一回收拢在JobTriggerPoolHelper入口，addTigger方法在选择两种不同策略的线程池后会继续执行后续的步骤，线程池分为fast和slow两种，目前只区分了job执行超时的jobId放入slow线程池，进行优先级的隔离。

下面是主要的触发Job逻辑，看注释也会一目了然，这里不做赘述。路由策略下面会讲到。

```
//trigger()
// load data
// sharding param
// processTrigger()
private static void processTrigger(XxlJobGroup group, XxlJobInfo jobInfo, int
finalFailRetryCount, TriggerTypeEnum triggerType, int index, int total){
    // param
    // 1、save log-id
    // 2、init trigger-param
    // 3、init address, 重要的调度机器策略（分为9种调度策略）
    routeAddressResult =
executorRouteStrategyEnum.getRouter().route(triggerParam,
group.getRegistryList()); // route就是选择路由的方式
    if (routeAddressResult.getCode() == ReturnT.SUCCESS_CODE) {
        address = routeAddressResult.getContent();
    }
    // 4、trigger remote executor, 重要的执行逻辑单元
    ReturnT<String> triggerResult = null;
    if (address != null) {
        triggerResult = runExecutor(triggerParam, address); // 执行器
    } else {
        triggerResult = new ReturnT<String>(ReturnT.FAIL_CODE, null);
    }
    // 5、collection trigger info
    // 6、save log trigger-info
}
```

得到routeAddressResult的路由策略存在很多种，下图展示了9种方式，顾名思义就是具体的路由策略，找到对应的执行机器的ip+port，返回address，具体的调度分配机器逻辑就完成了。



获取到具体的address后就转到具体的执行逻辑单元，下图进入runExecutor分为两个逻辑，1.获取ExecutorBiz 2. run（核心的执行Job逻辑）。首先我们先看下获取biz的逻辑，先从map里面获取，成功后直接返回，map中不存在则创建biz，然后成功createBiz后放入map，这里涉及到具体的RPC invoker逻辑。现着重看下init biz的核心逻辑

底层获取biz的方式可以看到就是通过RPC的invoker来获取对应的执行器，我们重新理解下，在调度前准备好具体映射的机器ip+port，调度的逻辑执行器通过RPC的调用来获取对应的objectName和methodName，得到普通的bean。在执行前得到了具体的远程机器的本地代理。

```
// initClient
initClient();
// newProxyInstance
return
Proxy.newProxyInstance(Thread.currentThread().getContextClassLoader(), new
Class[]{ iface },
    new InvocationHandler() {
        @Override
        public Object invoke (Object proxy, Method method, Object[]args) throws
Throwable {
            // method param
            // filter for generic
            // filter method like "Object.toString()"
            // address
            String finalAddress = address;
            if (finalAddress == null || finalAddress.trim().length() == 0) {
                if (invokerFactory != null && invokerFactory.getServiceRegistry()
!= null) {
                    // discovery
                    String serviceKey =
XxlRpcProviderFactory.makeServiceKey(className, version_);
                    TreeSet<String> addressSet =
invokerFactory.getServiceRegistry().discovery(serviceKey);
                    // load balance
                    if (addressSet == null || addressSet.size() == 0) {
                        // pass
                    } else if (addressSet.size() == 1) {
                        finalAddress = addressSet.first(); // 取第一个
                    } else {
                        finalAddress =
loadBalance.xxlRpcInvokerRouter.route(serviceKey, addressSet); // 路由选中
                    }
                }
            }
            // request, 构建请求
            XxlRpcRequest xxlRpcRequest = new XxlRpcRequest();
            xxlRpcRequest.setRequestId(UUID.randomUUID().toString());
            xxlRpcRequest.setCreateTime(System.currentTimeMillis());
            xxlRpcRequest.setAccessToken(accessToken);
            xxlRpcRequest.setClassName(className);
            xxlRpcRequest.setMethodName(methodName);
```

```

xxlRpcRequest.setParameterTypes(parameterTypes);
xxlRpcRequest.setParameters(parameters);
xxlRpcRequest.setVersion(version);

// send
if (CallType.SYNC == callType) {
    // future-response set
    XxlRpcFutureResponse futureResponse = new
XxlRpcFutureResponse(invokerFactory, xxlRpcRequest, null);
    try {
        // do invoke
        clientInstance.asyncSend(finalAddress, xxlRpcRequest);
        // future get
        XxlRpcResponse xxlRpcResponse = futureResponse.get(timeout,
TimeUnit.MILLISECONDS);
        if (xxlRpcResponse.getErrorMsg() != null) {
            throw new XxlRpcException(xxlRpcResponse.getErrorMsg());
        }
        return xxlRpcResponse.getResult(); //return RPC result, 这里就是
具体的beanName引用封装
    } catch (Exception e) {
        throw (e instanceof XxlRpcException) ? e : new
XxlRpcException(e);
    } finally {
        // future-response remove
        futureResponse.removeInvokerFuture();
    }
} else if (CallType.FUTURE == callType) {
    // future-response set
    XxlRpcFutureResponse futureResponse = new
XxlRpcFutureResponse(invokerFactory, xxlRpcRequest, null);
    try {
        // invoke future set
        XxlRpcInvokeFuture invokeFuture = new
XxlRpcInvokeFuture(futureResponse);
        XxlRpcInvokeFuture.setFuture(invokeFuture);

        // do invoke
        clientInstance.asyncSend(finalAddress, xxlRpcRequest);

        return null;
    } catch (Exception e) {
        // future-response remove
        futureResponse.removeInvokerFuture();
        throw (e instanceof XxlRpcException) ? e : new
XxlRpcException(e);
    }
} else if (CallType.CALLBACK == callType) {
    // get callback

```

```

        XxlRpcInvokeCallback finalInvokeCallback = invokeCallback;
        XxlRpcInvokeCallback threadInvokeCallback =
XxlRpcInvokeCallback.getCallback();
        if (threadInvokeCallback != null) {
            finalInvokeCallback = threadInvokeCallback;
        }
        if (finalInvokeCallback == null) {
            throw new XxlRpcException("xxl-rpc
XxlRpcInvokeCallback (CallType=" + CallType.CALLBACK.name() + ") cannot be
null.");
        }
        // future-response set
        XxlRpcFutureResponse futureResponse = new
XxlRpcFutureResponse(invokerFactory, xxlRpcRequest, finalInvokeCallback);
        try {
            clientInstance.asyncSend(finalAddress, xxlRpcRequest);
        } catch (Exception e) {
            // future-response remove
            futureResponse.removeInvokerFuture();
            throw (e instanceof XxlRpcException) ? e : new
XxlRpcException(e);
        }
        return null;
    } else if (CallType.ONEWAY == callType) {
        clientInstance.asyncSend(finalAddress, xxlRpcRequest);
        return null;
    } else {
        throw new XxlRpcException("xxl-rpc callType[" + callType + "]
invalid");
    }
}
});

```

然后到下面code的具体执行逻辑，实际上执行的具体逻辑在JobThread, 拉起thread并且把data推进queue中后，单个的thread来具体执行触发任务的逻辑

```

// replace thread (new or exists invalid)
if (jobThread == null) {
    jobThread = XxlJobExecutor.registJobThread(triggerParam.getJobId(),
jobHandler, removeOldReason);
}
// push data to queue
ReturnT<String> pushResult = jobThread.pushTriggerQueue(triggerParam);

```

下面我们去thread里看下具体的执行逻辑，FutureTask执行得到结果，Job的处理就基本结束了

```

// handler.init();

```

```

handler.init();

triggerParam = triggerQueue.poll(3L, TimeUnit.SECONDS);
if (triggerParam.getExecutorTimeout() > 0) {
    // limit timeout
    Thread futureThread = null;
    final TriggerParam triggerParamTmp = triggerParam;
    FutureTask<ReturnT<String>> futureTask = new FutureTask<ReturnT<String>>(new
Callable<ReturnT<String>>() {
        @Override
        public ReturnT<String> call() throws Exception {
            return handler.execute(triggerParamTmp.getExecutorParams()); // 执行，具体
的handler来处理
        }
    });
    futureThread = new Thread(futureTask);
    futureThread.start();
    executeResult = futureTask.get(triggerParam.getExecutorTimeout(),
TimeUnit.SECONDS);
} else {
    // just execute
    executeResult = handler.execute(triggerParam.getExecutorParams());
}

```

执行完任务之后，Thread会启动destory来执行结束任务后的处理逻辑，相当于开放了扩展的逻辑

```

// callback trigger request in queue
while(triggerQueue !=null && triggerQueue.size()>0){
    TriggerParam triggerParam = triggerQueue.poll();
    if (triggerParam!=null) {
        // is killed
        ReturnT<String> stopResult = new ReturnT<String>(ReturnT.FAIL_CODE,
stopReason + " [job not executed, in the job queue, killed.]");
        TriggerCallbackThread.pushCallBack(new
HandleCallbackParam(triggerParam.getLogId(), triggerParam.getLogDateTime(),
stopResult));
    }
}

// destroy, 执行对应的destory逻辑
try {
    handler.destroy();
} catch (Throwable e) {
    logger.error(e.getMessage(), e);
}

```

对应到具体的触发handler就是下面这个逻辑

```

// aop会被注册到handler上，init和destory都会被注册上去

```

```

@XxlJob(value = "demoJobHandler2", init = "init", destroy = "destroy")
public ReturnT<String> demoJobHandler2(String param) throws Exception {
    XxlJobLogger.log("XXL-JOB, Hello World.");
    return ReturnT.SUCCESS;
}

public void init(){
    logger.info("init");
}

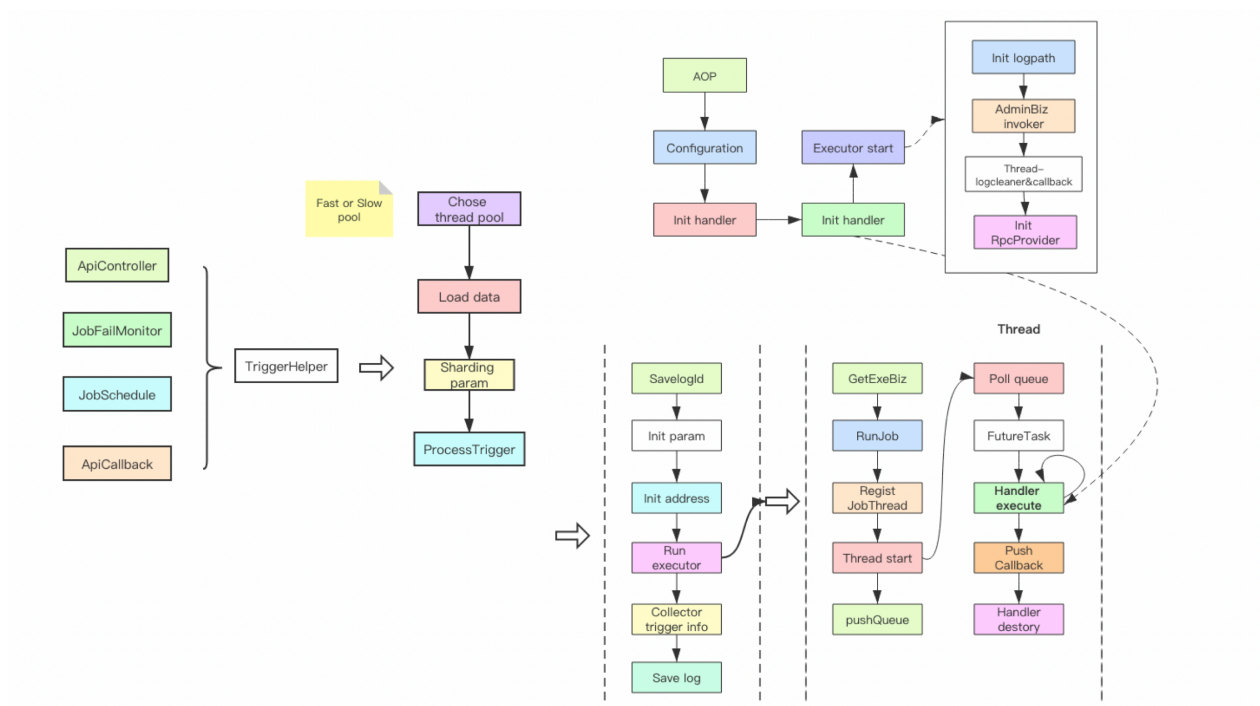
public void destroy(){
    logger.info("destory");
}

// 注册的逻辑
// registry jobhandler
registJobHandler(name, new MethodJobHandler(bean, method, initMethod,
destroyMethod));

```

架构图

架构图如下，显示了大致的模块和逻辑，从JobHandler的注册，到RPC的server的init，再到Job触发的方式和路由策略，以及Job的调度和执行Invoker。整体逻辑比较清晰



我们分析下该组件的核心要素

1. Job注册器（AOP，包括init和destory方法透出）

2. 路由策略，选中address机器来执行
3. RPC调用，得到具体执行机器上的beanName和method返回
4. JobThread, push data to queue, 然后执行具体的Handler绑定的执行方法来执行
5. 日志采集器和执行记录

设计规约

- 分布式调度(去中心化的服务， backup机制) 通过zk， 分布式锁来在分布式环境下保证单台机器执行
- 数据持久化（内存/数据库）
- 策略和调度解耦
- 任务可恢复可重试， 数据产出
- 作业注册中心
- 监控报警
- HA (高可用)
- 弹性扩容
- 动态分片（支持任务分片）

其他的调度框架

其他的调度框架

只考虑分布式下的环境中，有如下的调度框架

- Quartz（利用数据库锁来实现分布式环境下的并发控制） [Quartz应用与集群原理分析](#)
- Elastic-Job
- LTS
- SchedulerX

好的调度组件设计是什么样的？

好的调度组件设计是什么样的？

任务调度是业务发展中不可或缺的一部分，优秀的调度组件应该是什么样子呢？我们来设想一下，我们现存的调度平台的优点和痛点是什么？理想化的组件，需要满足下面的设计

- HA（包括存储和可用率）
- 业务逻辑和组件高度解耦合，调度策略多样化
- 易于扩展和维护
- 易于上手维护，配置简单