# C#.net 简单任务调度平台 使用demo

[收藏](javascript:void(0);)

[车江毅](https://my.oschina.net/chejiangyi/home)

* 发表于 1年前

* 阅读 559

* 收藏 0

* 点赞 0

* [评论 0](https://my.oschina.net/chejiangyi/blog/484642#comment-list)

*摘要: C#.net 简单任务调度平台 使用demo*

using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Dyd.BaseService.TaskManager.Demo

{

    /// <summary>

    /// 任务调度平台 任务插件的写法demo及说明

    /// </summary>

    public class DemoTask : XXF.BaseService.TaskManager.BaseDllTask

    {

        /// <summary>

        /// 任务调度平台根据发布的任务时间配置，定时回调运行方法

        /// 开发人员的任务插件必须要重载并该方法

        /// </summary>

        public override void Run()

        {

            /\*

             \* this.OpenOperator 用于任务调度平台提供给第三方使用的所有api接口封装

             \*/

            /\*获取当前任务dll安装目录\*/

            this.OpenOperator.GetTaskInstallDirectory();

            /\*打印一条日志到任务调度平台,因为日志会存到平台数据库，所以日志要精简，对任务出错时有分析价值【注意：不要频繁打印无用的，非必要的，对分析无价值的日志信息】\*/

            this.OpenOperator.Log("这里打印一条日志到任务调度平台");

            /\*打印一条错误到任务调度平台,因为日志会存到平台数据库，所以日志要精简，对任务出错时有分析价值【注意：不要频繁打印无用的，非必要的，对分析无价值的日志信息】

             \*后续任务会有增加优先级区分，根据任务的优先级,错误的出现频率等,错误日志会定期推送到开发者邮箱和短信\*/

            this.OpenOperator.Error("这里打印一条错误日志到任务调度平台", new Exception("错误msg信息"));

            /\*从数据库获取任务的临时数据，临时数据以jason的形式保存在数据库里面，便于任务上下文的恢复和信息传递【注意：不应用于"频繁的"存储"大量的"临时数据，会操作网络耗时和数据库性能差】\*/

            var databasetempinfo = this.OpenOperator.GetDataBaseTempData<DemoTaskDatabaseTempInfo>();

            if (databasetempinfo == null)//若任务第一次运行，可能没有临时数据。当然也可以在发布任务的时候配置临时数据也可。

            {

                databasetempinfo = new DemoTaskDatabaseTempInfo();

                databasetempinfo.LastLogID = 0;

            }

            /\*将任务的临时数据持久化到数据库中，临时数据以json的形式保存在数据库里面，便于任务上下文的恢复和信息传递【注意：不应用于"频繁的"存储"大量的"临时数据，会操作网络耗时和数据库性能差】

              若临时数据用于下一次使用，必须要执行此方法，否则下次无法获取【注意:执行此方法,当前临时数据有可能被重置为null，便于内存资源释放】\*/

            this.OpenOperator.SaveDataBaseTempData(databasetempinfo);

            /\*从本地安装目录中获取任务的临时数据，临时数据以jason的形式保存在本地，便于任务上下文的恢复和信息传递【注意：本地临时数据一般用于保存"大量的"临时数据】\*/

            var localtempinfo = this.OpenOperator.GetLocalTempData<DemoTaskLocalTempInfo>();

            if (localtempinfo == null)//若任务第一次运行，可能没有临时数据。当然也可以在发布任务的时候上传临时数据json至安装压缩包中也可。

            {

                localtempinfo = new DemoTaskLocalTempInfo();

                localtempinfo.file = new byte[0];

            }

            /\*将任务的临时数据持久化到本地安装目录中，临时数据以json的形式保存在本地安装目录里面，便于任务上下文的恢复和信息传递【注意：本地临时数据一般用于保存"大量的"临时数据】

              若临时数据用于下一次使用，必须要执行此方法，否则下次可能无法获取【注意:执行此方法,当前临时数据有可能被重置为null，便于内存资源释放】\*/

            this.OpenOperator.SaveLocalTempData(localtempinfo);

            {

                string msg = "执行业务中...."+ this.AppConfig["sendmailhost"];

                Debug.WriteLine(msg);

                System.IO.File.AppendAllText(this.OpenOperator.GetTaskInstallDirectory()+"业务.txt", msg);

            }

        }

        /// <summary>

        /// 开发人员自测运行入口

        /// 需要将项目配置为->控制台应用程序，写好Program类和Main入口函数

        /// </summary>

        public override void TestRun()

        {

            /\*测试环境下任务的配置信息需要手工填写,正式环境下需要配置在任务配置中心里面\*/

            this.AppConfig = new XXF.BaseService.TaskManager.SystemRuntime.TaskAppConfigInfo();

            this.AppConfig.Add("sendmailhost", "smtp.163.com");

            this.AppConfig.Add("sendmailname", "fengyeguigui@163.com");

            this.AppConfig.Add("password", "\*\*\*\*\*\*");

            base.TestRun();

        }

    }

    /// <summary>

    /// 任务调度平台之临时数据信息,用于任务上下文的信息传递。

    /// 将会以json形式保存在任务调度平台数据库中,便于下一次回调运行时恢复并使用。【注意：不应用于"频繁的"存储"大量的"临时数据，会操作网络耗时和数据库性能差】

    /// </summary>

    public class DemoTaskDatabaseTempInfo

    {

        public int LastLogID { get; set; }

    }

    /// <summary>

    /// 任务调度平台之临时数据信息,用于任务上下文的信息传递。

    /// 将会以json形式保存在任务调度平台本地安装文件夹中,便于下一次回调运行时恢复并使用。【注意：本地临时数据一般用于保存"大量的"临时数据】

    /// </summary>

    public class DemoTaskLocalTempInfo

    {

        public byte[] file { get; set; }

    }

}

项目开源地址:<http://git.oschina.net/chejiangyi/Dyd.BaseService.TaskManager>

/*\*作者:车江毅\**/

# .net 分布式锁实现

[收藏](javascript:void(0);)

[车江毅](https://my.oschina.net/chejiangyi/home)

* 发表于 1年前

* 阅读 828

* 收藏 5

* 点赞 0

* [评论 0](https://my.oschina.net/chejiangyi/blog/511291#comment-list)

*摘要: .net分布式锁，包括redis分布式锁和zookeeper分布式锁的.net实现。 分布式锁在解决分布式环境下的业务一致性是非常有用的。*

**分布式锁**

经常用于在解决分布式环境下的业务一致性和协调分布式环境。

实际业务场景中，比如说解决并发一瞬间的重复下单，重复确认收货，重复发现金券等。

使用分布式锁的场景一般不能太多。

开源地址：<http://git.oschina.net/chejiangyi/XXF.BaseService.DistributedLock>

**开源相关群: .net 开源基础服务 238543768**

这里整理了C#.net关于redis分布式锁和zookeeper分布式锁的实现，仅用于研究。（可能有bug）

**采用ServiceStack.Redis实现Redis分布式锁**

 /\*

 \* Redis分布式锁

 \* 采用ServiceStack.Redis实现的Redis分布式锁

 \* 详情可阅读其开源代码

 \* 备注：不同版本的 ServiceStack.Redis 实现reidslock机制不同 xxf里面默认使用2.2版本

 \*/    public class RedisDistributedLock : BaseRedisDistributedLock

    {

        private ServiceStack.Redis.RedisLock \_lock;

        private RedisClient \_client;

        public RedisDistributedLock(string redisserver, string key)

            : base(redisserver, key)

        {

        }

        public override LockResult TryGetDistributedLock(TimeSpan? getlockTimeOut, TimeSpan? taskrunTimeOut)

        {

            if (lockresult == LockResult.Success)

                throw new DistributedLockException("检测到当前锁已获取");

            \_client = DistributedLockConfig.GetRedisPoolClient(redisserver).GetClient();

            /\*             \* 阅读源码发现当其获取锁后,redis连接资源会一直占用,知道获取锁的资源释放后,连接才会跳出，可能会导致连接池资源的浪费。             \*/            try            {

                this.\_lock = new ServiceStack.Redis.RedisLock(\_client, key, getlockTimeOut);

                lockresult =  LockResult.Success;

            }

            catch (Exception exp)

            {

                XXF.Log.ErrorLog.Write(string.Format("redis分布式尝试锁系统级别严重异常,redisserver:{0}", redisserver.NullToEmpty()), exp);

                lockresult = LockResult.LockSystemExceptionFailure;

            }

            return lockresult;

        }

        public override void Dispose()

        {

            try            {

                if (this.\_lock != null)

                    this.\_lock.Dispose();

                if (\_client != null)

                    this.\_client.Dispose();

            }

            catch (Exception exp)

            {

                XXF.Log.ErrorLog.Write(string.Format("redis分布式尝试锁释放严重异常,redisserver:{0}", redisserver.NullToEmpty()), exp);

            }

        }

    }

**来自网络的java实现Redis分布式锁（C#版）**

/\*

\* Redis分布式锁

\* 采用网络上java实现的Redis分布式锁

\* 参考 http://www.blogjava.net/hello-yun/archive/2014/01/15/408988.html

\* 详情可阅读其开源代码

\*/    public class RedisDistributedLockFromJava : BaseRedisDistributedLock

    {

        public RedisDistributedLockFromJava(string redisserver, string key)

            : base(redisserver, key)

        {

        }

        public override LockResult TryGetDistributedLock(TimeSpan? getlockTimeOut, TimeSpan? taskrunTimeOut)

        {

            if (lockresult == LockResult.Success)

                throw new DistributedLockException("检测到当前锁已获取");

            try            {

                // 1. 通过SETNX试图获取一个lock                string @lock = key;

                long taskexpiredMilliseconds = (taskrunTimeOut != null ? (long)taskrunTimeOut.Value.TotalMilliseconds : (long)DistributedLockConfig.MaxLockTaskRunTime);

                long getlockexpiredMilliseconds = (getlockTimeOut != null ? (long)getlockTimeOut.Value.TotalMilliseconds : 0);

                long hassleepMilliseconds = 0;

                while (true)

                {

                    using (var redisclient = DistributedLockConfig.GetRedisPoolClient(redisserver).GetClient())

                    {

                        long value = CurrentUnixTimeMillis() + taskexpiredMilliseconds + 1;

                        /\*Java以前版本都是用SetNX,但是这种是无法设置超时时间的,不是很理解为什么,                         \* 可能是因为原来的redis命令比较少导致的？现在用Add不知道效果如何.                         因对redis细节不了解，但个人怀疑若异常未释放锁经常发生，可能会导致内存逐步溢出\*/                        bool acquired = redisclient.Add<long>(@lock, value, TimeSpan.FromMilliseconds(taskexpiredMilliseconds + DistributedLockConfig.TaskLockDelayCleepUpTime));

                        //SETNX成功，则成功获取一个锁

                        if (acquired == true)

                        {

                            lockresult = LockResult.Success;

                        }

                        //SETNX失败，说明锁仍然被其他对象保持，检查其是否已经超时                        else                        {

                            var oldValueBytes = redisclient.Get(@lock);

                            //超时

                            if (oldValueBytes != null && BitConverter.ToInt64(oldValueBytes, 0) < CurrentUnixTimeMillis())

                            {

                                /\*此处虽然重设并获取锁,但是超时时间可能被覆盖,故重设超时时间;若有进程一直在尝试获取锁，那么锁存活时间应该被延迟\*/                                var getValueBytes = redisclient.GetSet(@lock, BitConverter.GetBytes(value));

                                var o1 = redisclient.ExpireEntryIn(@lock, TimeSpan.FromMilliseconds(taskexpiredMilliseconds + DistributedLockConfig.TaskLockDelayCleepUpTime));//这里如果程序异常终止，依然会有部分锁未释放的情况。                                // 获取锁成功                                if (getValueBytes == oldValueBytes)

                                {

                                    lockresult = LockResult.Success;

                                }

                                // 已被其他进程捷足先登了

                                else

                                {

                                    lockresult = LockResult.GetLockTimeOutFailure;

                                }

                            }

                            //未超时，则直接返回失败

                            else

                            {

                                lockresult = LockResult.GetLockTimeOutFailure;

                            }

                        }

                    }

                    //成功拿到锁

                    if (lockresult == LockResult.Success)

                        break;

                    //获取锁超时

                    if (hassleepMilliseconds >= getlockexpiredMilliseconds)

                    {

                        lockresult = LockResult.GetLockTimeOutFailure;

                        break;

                    }

                    //继续等待

                    System.Threading.Thread.Sleep(DistributedLockConfig.GetLockFailSleepTime);

                    hassleepMilliseconds += DistributedLockConfig.GetLockFailSleepTime;

                }

            }

            catch (Exception exp)

            {

                XXF.Log.ErrorLog.Write(string.Format("redis分布式尝试锁系统级别严重异常,redisserver:{0}", redisserver.NullToEmpty()), exp);

                lockresult = LockResult.LockSystemExceptionFailure;

            }

            return lockresult;

        }

        private long CurrentUnixTimeMillis()

        {

            return (long)(System.DateTime.UtcNow - new System.DateTime(1970, 1, 1, 0, 0, 0, System.DateTimeKind.Utc)).TotalMilliseconds;

        }

        public override void Dispose()

        {

            if (lockresult == LockResult.Success || lockresult == LockResult.LockSystemExceptionFailure)

            {

                try                {

                    long current = CurrentUnixTimeMillis();

                    using (var redisclient = DistributedLockConfig.GetRedisPoolClient(redisserver).GetClient())

                    {

                        var v = redisclient.Get(key);

                        if (v != null)

                        {

                            // 避免删除非自己获取得到的锁

                            if (current < BitConverter.ToInt64(v, 0))

                            {

                                redisclient.Del(key);

                            }

                        }

                    }

                }

                catch (Exception exp)

                {

                    XXF.Log.ErrorLog.Write(string.Format("redis分布式尝试锁释放严重异常,redisserver:{0}", redisserver.NullToEmpty()), exp);

                }

            }

        }

    }

**ServiceStack.Redis内部实现版本（较旧）**

 /\*

  \* Redis分布式锁

 \* 采用ServiceStack.Redis实现的Redis分布式锁

 \* 详情可阅读其开源代码

 \* 备注：不同版本的 ServiceStack.Redis 实现reidslock机制不同

  \* 拷贝自网络开源代码 较旧的实现版本

  \*/    public class RedisDistributedLockFromServiceStack : BaseRedisDistributedLock

    {

        public RedisDistributedLockFromServiceStack(string redisserver, string key)

            : base(redisserver, key)

        {

        }

        public override LockResult TryGetDistributedLock(TimeSpan? getlockTimeOut, TimeSpan? taskrunTimeOut)

        {

            if (lockresult == LockResult.Success)

                throw new DistributedLockException("检测到当前锁已获取");

            try

            {

                using (var redisClient = DistributedLockConfig.GetRedisPoolClient(redisserver).GetClient())

                {

                    ExecExtensions.RetryUntilTrue(

                             () =>

                             {

                                 //This pattern is taken from the redis command for SETNX http://redis.io/commands/setnx

                                  //Calculate a unix time for when the lock should expire                                 TimeSpan realSpan = taskrunTimeOut ?? TimeSpan.FromMilliseconds(DistributedLockConfig.MaxLockTaskRunTime); //new TimeSpan(365, 0, 0, 0); //if nothing is passed in the timeout hold for a year                                 DateTime expireTime = DateTime.UtcNow.Add(realSpan);

                                 string lockString = (expireTime.ToUnixTimeMs() + 1).ToString();

                                 //Try to set the lock, if it does not exist this will succeed and the lock is obtained                                 var nx = redisClient.SetEntryIfNotExists(key, lockString);

                                 if (nx)

                                 {

                                     lockresult = LockResult.Success;

                                     return true;

                                 }

                                 //If we've gotten here then a key for the lock is present. This could be because the lock is                                 //correctly acquired or it could be because a client that had acquired the lock crashed (or didn't release it properly).                                 //Therefore we need to get the value of the lock to see when it should expire

                                  redisClient.Watch(key);

                                 string lockExpireString = redisClient.Get<string>(key);

                                 long lockExpireTime;

                                 if (!long.TryParse(lockExpireString, out lockExpireTime))

                                 {

                                     redisClient.UnWatch();  // since the client is scoped externally                                     lockresult = LockResult.GetLockTimeOutFailure;

                                     return false;

                                 }

                                 //If the expire time is greater than the current time then we can't let the lock go yet                                 if (lockExpireTime > DateTime.UtcNow.ToUnixTimeMs())

                                 {

                                     redisClient.UnWatch();  // since the client is scoped externally                                     lockresult = LockResult.GetLockTimeOutFailure;

                                     return false;

                                 }

                                 //If the expire time is less than the current time then it wasn't released properly and we can attempt to                                  //acquire the lock. The above call to Watch(\_lockKey) enrolled the key in monitoring, so if it changes                                 //before we call Commit() below, the Commit will fail and return false, which means that another thread                                  //was able to acquire the lock before we finished processing.                                 using (var trans = redisClient.CreateTransaction()) // we started the "Watch" above; this tx will succeed if the value has not moved                                  {

                                     trans.QueueCommand(r => r.Set(key, lockString));

                                     //return trans.Commit(); //returns false if Transaction failed                                     var t = trans.Commit();

                                     if (t == false)

                                         lockresult = LockResult.GetLockTimeOutFailure;

                                     else

                                     lockresult = LockResult.Success;

                                     return t;

                                 }

                             },

                             getlockTimeOut

                         );

                }

            }

            catch (Exception exp)

            {

                XXF.Log.ErrorLog.Write(string.Format("redis分布式尝试锁系统级别严重异常,redisserver:{0}", redisserver.NullToEmpty()), exp);

                lockresult = LockResult.LockSystemExceptionFailure;

            }

            return lockresult;

        }

        public override void Dispose()

        {

            try            {

                using (var redisClient = DistributedLockConfig.GetRedisPoolClient(redisserver).GetClient())

                {

                    redisClient.Remove(key);

                }

            }

            catch (Exception exp)

            {

                XXF.Log.ErrorLog.Write(string.Format("redis分布式尝试锁释放严重异常,redisserver:{0}", redisserver.NullToEmpty()), exp);

            }

        }

    }

**Zookeeper 版本实现分布式锁**

/\*  \* 来源java网络源码的zookeeper分布式锁实现（目前仅翻译并简单测试ok，未来集成入sdk）

\* 备注:    共享锁在同一个进程中很容易实现，但是在跨进程或者在不同 Server 之间就不好实现了。Zookeeper 却很容易实现这个功能，实现方式也是需要获得锁的 Server 创建一个 EPHEMERAL\_SEQUENTIAL 目录节点，    然后调用 getChildren方法获取当前的目录节点列表中最小的目录节点是不是就是自己创建的目录节点，如果正是自己创建的，那么它就获得了这个锁，    如果不是那么它就调用 exists(String path, boolean watch) 方法并监控 Zookeeper 上目录节点列表的变化，一直到自己创建的节点是列表中最小编号的目录节点，    从而获得锁，释放锁很简单，只要删除前面它自己所创建的目录节点就行了。

\*/    public class ZooKeeprDistributedLockFromJava : IWatcher

    {

        private ZooKeeper zk;

        private string root = "/locks"; //根

        private string lockName; //竞争资源的标志

        private string waitNode; //等待前一个锁

        private string myZnode; //当前锁

        //private CountDownLatch latch; //计数器

        private AutoResetEvent autoevent;

        private TimeSpan sessionTimeout = TimeSpan.FromMilliseconds(30000);

        private IList<Exception> exception = new List<Exception>();

        /// <summary>

        /// 创建分布式锁,使用前请确认config配置的zookeeper服务可用 </summary>

        /// <param name="config"> 127.0.0.1:2181 </param>

        /// <param name="lockName"> 竞争资源标志,lockName中不能包含单词lock </param>

        public ZooKeeprDistributedLockFromJava(string config, string lockName)

        {

            this.lockName = lockName;

            // 创建一个与服务器的连接

            try            {

                zk = new ZooKeeper(config, sessionTimeout, this);

                var stat = zk.Exists(root, false);

                if (stat == null)

                {

                    // 创建根节点

                    zk.Create(root, new byte[0], Ids.OPEN\_ACL\_UNSAFE, CreateMode.Persistent);

                }

            }

            catch (KeeperException e)

            {

                throw e;

            }

        }

        /// <summary>

        /// zookeeper节点的监视器

        /// </summary>

        public virtual void Process(WatchedEvent @event)

        {

            if (this.autoevent != null)

            {

                this.autoevent.Set();

            }

        }

        public virtual bool tryLock()

        {

            try            {

                string splitStr = "\_lock\_";

                if (lockName.Contains(splitStr))

                {

                    //throw new LockException("lockName can not contains \\u000B");                }

                //创建临时子节点

                myZnode = zk.Create(root + "/" + lockName + splitStr, new byte[0], Ids.OPEN\_ACL\_UNSAFE, CreateMode.EphemeralSequential);

                Console.WriteLine(myZnode + " is created ");

                //取出所有子节点

                IList<string> subNodes = zk.GetChildren(root, false);

                //取出所有lockName的锁

                IList<string> lockObjNodes = new List<string>();

                foreach (string node in subNodes)

                {

                    if (node.StartsWith(lockName))

                    {

                        lockObjNodes.Add(node);

                    }

                }

                Array alockObjNodes = lockObjNodes.ToArray();

                Array.Sort(alockObjNodes);

                Console.WriteLine(myZnode + "==" + lockObjNodes[0]);

                if (myZnode.Equals(root + "/" + lockObjNodes[0]))

                {

                    //如果是最小的节点,则表示取得锁

                    return true;

                }

                //如果不是最小的节点，找到比自己小1的节点

                string subMyZnode = myZnode.Substring(myZnode.LastIndexOf("/", StringComparison.Ordinal) + 1);

                waitNode = lockObjNodes[Array.BinarySearch(alockObjNodes, subMyZnode) - 1];

            }

            catch (KeeperException e)

            {

                throw e;

            }

            return false;

        }

        public virtual bool tryLock(TimeSpan time)

        {

            try            {

                if (this.tryLock())

                {

                    return true;

                }

                return waitForLock(waitNode, time);

            }

            catch (KeeperException e)

            {

                throw e;

            }

            return false;

        }

        private bool waitForLock(string lower, TimeSpan waitTime)

        {

            var stat = zk.Exists(root + "/" + lower, true);

            //判断比自己小一个数的节点是否存在,如果不存在则无需等待锁,同时注册监听

            if (stat != null)

            {

                Console.WriteLine("Thread " + System.Threading.Thread.CurrentThread.Name + " waiting for " + root + "/" + lower);

                autoevent = new AutoResetEvent(false);

                bool r = autoevent.WaitOne(waitTime);

                autoevent.Dispose();

                autoevent = null;

                return r;

            }

            else                return true;

        }

        public virtual void unlock()

        {

            try            {

                Console.WriteLine("unlock " + myZnode);

                zk.Delete(myZnode, -1);

                myZnode = null;

                zk.Dispose();

            }

            catch (KeeperException e)

            {

                throw e;

            }

        }

    }

以上代码仅做参考，未压测。

# 【开源】.net 分布式架构之任务调度平台

[收藏](javascript:void(0);)

[车江毅](https://my.oschina.net/chejiangyi/home)

* 发表于 1年前

* 阅读 2434

* 收藏 13

* 点赞 2

* [评论 3](https://my.oschina.net/chejiangyi/blog/484635#comment-list)

*摘要: .net 分布式任务调度平台，用于.net dll,exe的任务的挂载，任务的隔离，调度执行，访问权限控制，监控，管理，日志，错误预警，性能分析等。*

开源地址：<http://git.oschina.net/chejiangyi/Dyd.BaseService.TaskManager>

**.net 任务调度平台**

用于.net dll,exe的任务的挂载，任务的隔离，调度执行，访问权限控制，监控，管理，日志，错误预警，性能分析等。  
  
1) 平台基于quartz.net进行任务调度功能开发，采用C#代码编写, 支持corn表达式和第三方自定义的corn表达式扩展。  
  
2) 架构以插件形式开发，具有良好的功能扩展性，稳定性，简单性，便于第三方开发人员进一步进行功能扩展。  
  
3) 支持多节点集群，便于集群服务器的资源有效分配，任务的相互隔离。  
  
4) 支持邮件形式的错误预警，便于运维及时处理任务异常等。

**开源相关群: .net 开源基础服务 238543768**

**任务demo**

using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Dyd.BaseService.TaskManager.Demo

{

    /// <summary>

    /// 任务调度平台 任务插件的写法demo及说明

    /// </summary>

    public class DemoTask : XXF.BaseService.TaskManager.BaseDllTask

    {

        /// <summary>

        /// 任务调度平台根据发布的任务时间配置，定时回调运行方法

        /// 开发人员的任务插件必须要重载并该方法

        /// </summary>

        public override void Run()

        {

            /\*

             \* this.OpenOperator 用于任务调度平台提供给第三方使用的所有api接口封装

             \*/

            /\*获取当前任务dll安装目录\*/

            this.OpenOperator.GetTaskInstallDirectory();

            /\*打印一条日志到任务调度平台,因为日志会存到平台数据库，所以日志要精简，对任务出错时有分析价值【注意：不要频繁打印无用的，非必要的，对分析无价值的日志信息】\*/

            this.OpenOperator.Log("这里打印一条日志到任务调度平台");

            /\*打印一条错误到任务调度平台,因为日志会存到平台数据库，所以日志要精简，对任务出错时有分析价值【注意：不要频繁打印无用的，非必要的，对分析无价值的日志信息】

             \*后续任务会有增加优先级区分，根据任务的优先级,错误的出现频率等,错误日志会定期推送到开发者邮箱和短信\*/

            this.OpenOperator.Error("这里打印一条错误日志到任务调度平台", new Exception("错误msg信息"));

            /\*从数据库获取任务的临时数据，临时数据以jason的形式保存在数据库里面，便于任务上下文的恢复和信息传递【注意：不应用于"频繁的"存储"大量的"临时数据，会操作网络耗时和数据库性能差】\*/

            var databasetempinfo = this.OpenOperator.GetDataBaseTempData<DemoTaskDatabaseTempInfo>();

            if (databasetempinfo == null)//若任务第一次运行，可能没有临时数据。当然也可以在发布任务的时候配置临时数据也可。

            {

                databasetempinfo = new DemoTaskDatabaseTempInfo();

                databasetempinfo.LastLogID = 0;

            }

            /\*将任务的临时数据持久化到数据库中，临时数据以json的形式保存在数据库里面，便于任务上下文的恢复和信息传递【注意：不应用于"频繁的"存储"大量的"临时数据，会操作网络耗时和数据库性能差】

              若临时数据用于下一次使用，必须要执行此方法，否则下次无法获取【注意:执行此方法,当前临时数据有可能被重置为null，便于内存资源释放】\*/

            this.OpenOperator.SaveDataBaseTempData(databasetempinfo);

            /\*从本地安装目录中获取任务的临时数据，临时数据以jason的形式保存在本地，便于任务上下文的恢复和信息传递【注意：本地临时数据一般用于保存"大量的"临时数据】\*/

            var localtempinfo = this.OpenOperator.GetLocalTempData<DemoTaskLocalTempInfo>();

            if (localtempinfo == null)//若任务第一次运行，可能没有临时数据。当然也可以在发布任务的时候上传临时数据json至安装压缩包中也可。

            {

                localtempinfo = new DemoTaskLocalTempInfo();

                localtempinfo.file = new byte[0];

            }

            /\*将任务的临时数据持久化到本地安装目录中，临时数据以json的形式保存在本地安装目录里面，便于任务上下文的恢复和信息传递【注意：本地临时数据一般用于保存"大量的"临时数据】

              若临时数据用于下一次使用，必须要执行此方法，否则下次可能无法获取【注意:执行此方法,当前临时数据有可能被重置为null，便于内存资源释放】\*/

            this.OpenOperator.SaveLocalTempData(localtempinfo);

            {

                string msg = "执行业务中...."+ this.AppConfig["sendmailhost"];

                Debug.WriteLine(msg);

                System.IO.File.AppendAllText(this.OpenOperator.GetTaskInstallDirectory()+"业务.txt", msg);

            }

        }

        /// <summary>

        /// 开发人员自测运行入口

        /// 需要将项目配置为->控制台应用程序，写好Program类和Main入口函数

        /// </summary>

        public override void TestRun()

        {

            /\*测试环境下任务的配置信息需要手工填写,正式环境下需要配置在任务配置中心里面\*/

            this.AppConfig = new XXF.BaseService.TaskManager.SystemRuntime.TaskAppConfigInfo();

            this.AppConfig.Add("sendmailhost", "smtp.163.com");

            this.AppConfig.Add("sendmailname", "fengyeguigui@163.com");

            this.AppConfig.Add("password", "\*\*\*\*\*\*");

            base.TestRun();

        }

    }

    /// <summary>

    /// 任务调度平台之临时数据信息,用于任务上下文的信息传递。

    /// 将会以json形式保存在任务调度平台数据库中,便于下一次回调运行时恢复并使用。【注意：不应用于"频繁的"存储"大量的"临时数据，会操作网络耗时和数据库性能差】

    /// </summary>

    public class DemoTaskDatabaseTempInfo

    {

        public int LastLogID { get; set; }

    }

    /// <summary>

    /// 任务调度平台之临时数据信息,用于任务上下文的信息传递。

    /// 将会以json形式保存在任务调度平台本地安装文件夹中,便于下一次回调运行时恢复并使用。【注意：本地临时数据一般用于保存"大量的"临时数据】

    /// </summary>

    public class DemoTaskLocalTempInfo

    {

        public byte[] file { get; set; }

    }

}

**web后端部分截图及安装**

# 【开源】.net 分布式架构之监控平台

[收藏](javascript:void(0);)

[车江毅](https://my.oschina.net/chejiangyi/home)

* 发表于 1年前

* 阅读 5451

* 收藏 145

* 点赞 6

* [评论 21](https://my.oschina.net/chejiangyi/blog/510655#comment-list)

*摘要: .net 分布式集群监控平台,用于集群的性能监控,应用耗时监控管理，统一日志管理等多维度的性能监控分析。*

开源地址:<http://git.oschina.net/chejiangyi/Dyd.BaseService.Monitor>

**.net 简单监控平台**，用于集群的性能监控,应用耗时监控管理，统一日志管理等多维度的性能监控分析。  
  
**集群的性能监控**:（需要服务器部署监控节点）  
 1. 目前仅支持windows服务器监控,  
 2. 支持windows自带的性能分析器所有配置项监控，及预警的定制。  
 3. 支持自定义的插件扩展服务器采集dll进行各类性能采集及预警的定制。  
 4. 支持性能采集快照。  
 5. 支持cpu,内存，磁盘读写，网络上传下载，iis请求这些基础的服务器性能图展示。  
  
**应用耗时监控管理:**（需要应用在api和sql层嵌入sdk进行拦截）  
  1. 支持sql的耗时（均值，最大值，最小值，查询次数）性能分析及性能图展示。  
  2. 支持api接口的耗时（均值，最大值，最小值，查询次数）性能分析及性能图展示。  
  3. 支持api与api内部sql的对照参照分析。  
  4. 问题sql：未进行参数化sql的分析列表。  
  
**统一日志管理**：（需要应用集成sdk）  
  1. 整个平台的错误日志集中管理及预警邮件发送。  
  2. 整个平台的普通日志集中管理。  
  
**监控平台任务**:(需要将"任务"挂载在"任务调度平台"开源项目， [.net 任务调度平台](http://my.oschina.net/u/2379842/blog/484635))  
  1. 统计SQLHASH和URL的对照  
  2. 统计服务器监控信息  
  3. 自动创表  
  4. 统计SQLHASH对照和SQL执行次数  
  5. SQL性能监控统计  
  6. API性能监控统计  
  7. 耗时性能预警任务  
  8. 错误频率预警任务  
  9. 错误邮件发送任务  
  10.集群性能预警任务  
  
备注：  
每个公司都有特定的平台统一监控的需求及改进方案，所以开源的监控平台只是一种参考，并不适用于不同行业不同的项目。  
希望此开源项目能带来更多思路,成为同类优秀项目的起点。另外项目文档及安装资料目前未整理全面，后续完善。  
  
（项目允许用于第三方公司进行二次开发且在公司内部使用，但禁止以盈利为目的进行商业行为）  
  
未来构想：  
1. 底层采用nosql等其他存储进行大容量数据采集及分析。  
  
  
备注：  
1. 监控平台安装说明里面的文件是内部打包的文件，里面的xxf和开源的xxf是不一样的。若遇到xxf命名空间问题，请在编译的时候使用"引用"文件夹下面的xxf，那个xxf是为监控平台定制的阉割分离后的xxf。  
  
2. 监控平台登录集成了内部的crm权限系统，所以第三方使用者需修改web站点的登录代码。

**开源相关群: .net 开源基础服务 238543768**

有图有真相：

# 【开源】.net 分布式架构之业务消息队列

[收藏](javascript:void(0);)

[车江毅](https://my.oschina.net/chejiangyi/home)

* 发表于 1年前

* 阅读 3010

* 收藏 36

* 点赞 7

* [评论 3](https://my.oschina.net/chejiangyi/blog/515860#comment-list)

*摘要: .net 业务消息队列是应用于业务的解耦和分离，应具备分布式，高可靠性，高性能，高实时性，高稳定性，高扩展性等特性。大量的业务消息堆积能力;无单点故障及故障监控，异常提醒;.生产者端负载均衡，故障转移，故障自动恢复，并行消息插入;.消费者端负载均衡，故障保持，故障自动恢复，并行消息消费;消息高可靠性持久化，较高性能，较高实时性，高稳定性，高扩张性;支持99\*99个消息分区，单个消息分区单天支持近1亿的消息存储;消费者拉方式获取消息，在高并发，大量消息涌入的情况下，只要消费能力足够，不会有消息延迟，消息越多性能越好。*

**开源QQ群: .net 开源基础服务  238543768**

开源地址: <http://git.oschina.net/chejiangyi/Dyd.BusinessMQ>

## 业务消息队列 ##  
业务消息队列是应用于业务的解耦和分离，应具备分布式，高可靠性，高性能，高实时性，高稳定性，高扩展性等特性。  
  
## 优点: ##  
- 大量的业务消息堆积能力  
- 无单点故障及故障监控，异常提醒  
- 生产者端负载均衡，故障转移，故障自动恢复，并行消息插入。  
- 消费者端负载均衡，故障保持，故障自动恢复，并行消息消费。  
- 消息高可靠性持久化，较高性能，较高实时性，高稳定性，高扩展性。  
- 支持99\*99个消息分区，单个消息分区单天支持近1亿的消息存储。  
- 消费者拉方式获取消息，在高并发，大量消息涌入的情况下，只要消费能力足够，不会有消息延迟，消息越多性能越好。   
  
## 缺点： ##  
- 能保证消息顺序插入,保证相同分区的消息是顺序的（排除网络延迟），但是多个分区之间的可能是乱序的。  
- 消息并行消费或者多个分区并行消费或者负载均衡情况下的，消息消费顺序是乱序。  
  
## 缺点原因： ##  
- 消息的负载均衡是基于消息的分区存储，故多个分区之间的消息是乱序的，但是相同分区的消息是顺序的。  
- 消息的消费者负载均衡也是基于消息的分区进行均衡的，同时单个消费者订阅多个分区的情况下，也可并行进行消费。意味着不同分区的消息的消费是乱序的，但是相同分区的消息消费是顺序的。  
  
## 缺点解决方案： ##  
- 生产者自定义负载均衡算法，按照业务维度（用户，商户）等进行分区（多个用户之间可以消息乱序，单个用户的消息必须是顺序的），不同维度可以指向不同的分区，但是单个维度的消息是可以保证顺序的。  
- 本解决方案在故障的情况下，故障会移除某些故障节点，意味着故障节点会立即报错（当然也可自己指定故障节点进行转移，但是转移的节点消息会被提前消费，故障的消息会在恢复故障后重新消费，这样也会出现故障程度上的消息乱序消费）。  
- 本解决方案在线上无缝扩容和扩展性能方面也会有限制，看要具体的负载均衡算法，但是一般情况下，如果要扩容还是会进行部分消息迁移的情况。  
  
## 问答: ##  
### \*1.大量的业务消息堆积能力,如何实现？\* ###  
  每个分区表支持约1亿的消息存储，可以通过增加分区表进行扩容。消费者进行消息消费，内部仅保留某个分区上一次消费的指针，所以不会影响消费者。  
  消息持久化到磁盘，不会在内存驻留，理论上不影响内存。  
  
### \*2.无单点故障及故障监控，异常提醒？\* ###  
  故障一般会发生在redis，数据节点，管理中心，日志中心。  
  redis节点故障会影响消费者的消息消费响应及时度，一般延迟5s以内。不会影响消息消费速度和消息消费QPS  
  数据节点故障会影响生产者和消费者的消息，并造成消息暂时丢失（但是都是可恢复的，具体的看数据库的高可用做到什么程度）。  
  生产者端会无缝的进行节点移除，但是会默认1分钟重新尝试重连。消费者会持续报错至日志中，但是不会影响其他分区消费。  
  管理中心故障会影响生产者和消费者的心跳检测和新注册的生产者，消费者，但不会影响生产者和消费者具体的消息存储和发送接收。  
  日志中心故障不会影响生产者和消费者，但是影响日志的打印，日志中心故障会通知公司内部监控平台。  
  虽然故障不会影响线上已有的消息运行，但是还是会在高并发情况下出现性能问题，和系统稳定性，所以一旦发现要重视和及时处理。  
  
### \*3.生产者端负载均衡，故障转移，故障自动恢复，并行消息插入？\* ###  
  默认负载均衡采用多个分区顺序轮询插入，在并发情况下轮询插入是并行插入到不同分区的；某个数据节点出现故障，会移除相关数据节点的所有分区；  
  默认1分钟会重新载入故障分区进行重试。  
  
### \*4.消费者端负载均衡，故障保持，故障自动恢复，并行消息消费。\* ###  
  默认消费者端负载均衡是根据消费者订阅的分区进行的（一个消费者可以订阅多个分区，多个相同业务的消费者可以订阅多个不同分区进行负载）。  
  一个消费者订阅多个分区，这个消费者可以开启并行进行多分区消费。并行度=分区数，效果理论上最佳。  
  分区节点出现故障等，单个分区或者数据节点就会暂停消费，并通知日志中心打印错误日志。当故障恢复后，消费继续进行。  
  
### \*5.消息高可靠性持久化，较高性能，较高实时性，高稳定性，高稳定性。\* ###  
  消息传递到消息中心后，立即持久化到磁盘，故不会丢失消息。生产者可以采用多个分区进行并行插入，消费者可以采用并行进行消息消费，故理论上性能是可扩展无限量的。  
  消息是通过拉取的方式获取的，发送消息会由redis进行即时通知消费者拉取（即时消息默认会合并在500ms内redis通知消息），一般在20ms内消息会被消费掉。  
  批量拉消息的方式相对push的消息推送方式在高并发和大量消息处理的情况下，消息发送性能应该是更优的。  
  稳定性是基于数据库的稳定性和故障转移层面来确保的，扩展性体现在线上无缝的迁移和扩容。  
  
### \*6.支持9999个消息分区，单个消息分区单天支持近1亿的消息存储。\* ###  
  数据节点是01~99个，节点里面的表分区是01~99个，所以可以支持近1万个分区节点。单表的mqid最大应该是（1亿-1）条，应该满足一般的业务需求，  
  若不能满足，可以通过多个分区的方式扩容。  
  
### \*7.消费者拉方式获取消息，在高并发，大量消息涌入的情况下，只要消费能力足够，不会有消息延迟，消息越多性能越好。\* ###  
  push推消息的模式能保证更高的实时性，但是在大量消息的情况下，消息堆积的情况更严重，性能会有所影响。  
  pull拉消息的模式在保证消息实时性方面会略差，但是在大量消息涌入的情况下，批量拉消息效率更加。而且会将消息分发的负载转移到多个消费者端上。  
  
## 未来改进： ##  
1.  未来采用leveldb重写存储。  
1.  剥离broker服务用于支持相对可靠的消息服务。  
1.  消息完成标记本地缓存/持久化（或者存储redis）,每秒提交更新至数据库,消除频繁消费导致的瓶颈。

## 架构示意图 ##

## 使用demo示例 ##

 /// <summary>

        /// 发送消息

        /// </summary>

        /// <param name="msg"></param>

        public void SendMessageDemo(string msg)

        {

            //发送字符串示例

            var p = ProducterPoolHelper.GetPool(new BusinessMQConfig() {

                    ManageConnectString = "server=192.168.17.201;Initial

                    Catalog=dyd\_bs\_MQ\_manage;User ID=sa;Password=Xx~!@#;" },//管理中心数据库

                    "dyd.mytest3");//队列路径 .分隔,类似类的namespace,是队列的唯一标识，要提前告知运维在消息中心注册，方可使用。

            p.SendMessage(@"1");

            //发送对象示例

            /\* var obj = new message2 { text = "文字", num = 1 };

              var p = ProducterPoolHelper.GetPool(new BusinessMQConfig() {

                        ManageConnectString = "server=192.168.17.237;Initial

                        Catalog=dyd\_bs\_MQ\_manage;User ID=sa;Password=Xx~!@#;" },//管理中心数据库

                        "test.diayadian.obj");//队列路径 .分隔,类似类的namespace,是队列的唯一标识，要提前告知运维在消息中心注册，方可使用。

            p.SendMessage<message>(obj);

            \*/

        }

        private ConsumerProvider Consumer;

        /// <summary>

        /// 接收消息

        /// </summary>

        /// <param name="action"></param>

        public void ReceiveMessageDemo(Action<string> action)

        {

            if (Consumer == null)

            {

                Consumer = new ConsumerProvider();

                Consumer.Client = "dyd.mytest3.customer1";//clientid,接收消息的(消费者)唯一标示,一旦注册以后，不能更改，业务下线废弃后必须要告知运维，删除消费者注册。

                Consumer.ClientName = "客户端名称";//这个相对随意些，主要是用来自己识别的，要简短

                Consumer.Config = new BusinessMQConfig() { ManageConnectString =

                        "server=192.168.17.201;Initial Catalog=dyd\_bs\_MQ\_manage;User

                        ID=sa;Password=Xx~!@#;" };

                Consumer.MaxReceiveMQThread = 1;//并行处理的线程数,一般为1足够,若消息处理慢,又想并行消费,则考虑 正在使用的分区=并行处理线程数 为并行效率极端最优,但cpu消耗应该不小。

                Consumer.MQPath = "dyd.mytest3";//接收的队列要正确

                Consumer.PartitionIndexs = new List<int>() { 1, 2, 3,4, 5, 6, 7, 8 };//消费者订阅的分区顺序号,从1开始

                Consumer.RegisterReceiveMQListener<string>((r) =>

                {

                    /\*

                       \* 这些编写业务代码

                       \* 编写的时候要注意考虑，业务处理失败的情况。

                       \* 1.重试失败n次。

                       \* 2.重试还不行，则标记消息已被处理。然后跳过该消息处理，自己另外文档记录这种情况。

                       \* 消息被消费完毕，一定要调用MarkFinished，标记消息被消费完毕。

                       \*/

                    action.Invoke(r.ObjMsg);

                    r.MarkFinished();

                });

            }

        }

        /// <summary>

        /// 关闭消息订阅连接

        /// </summary>

        public void CloseReceiveMessage()

        {

            //注册消费者消息,消费者务必要在程序关闭后关掉（dispose）。否则导致异常终止,要人工等待连接超时后，方可重新注册。

            if (Consumer != null)

            {

                Consumer.Dispose();

                Consumer = null;

            }

        }

    }

**部分截图**

备注:.net开源的消息队列很少，特别是针对业务的高可靠性的消息队列；希望这个开源的消息队列，能够为.net领域带来更多解决方案，更多的思路和架构设计；同时也希望了解消息队列的人能够给于这个解决方案更多的建议和完善意见。

作者：车江毅

.net 大型分布式电子商务架构说明

[收藏](javascript:void(0);)

[车江毅](https://my.oschina.net/chejiangyi/home)

* 发表于 1年前

* 阅读 9493

* 收藏 278

* 点赞 41

* [评论 29](https://my.oschina.net/chejiangyi/blog/521950#comment-list)

*摘要: 构建具备高可用，高扩展性，高性能，能承载高并发，大流量的.net 分布式电子商务平台的架构说明。 其中包含基础框架沉淀,分库分表，基础服务架构(消息队列，任务调度，分布式缓存，配置中心，服务中心，监控平台等)，云建设，分布式操作系统搭建。*

# .net大型分布式电子商务架构说明

背景

构建具备高可用，高扩展性，高性能，能承载高并发，大流量的分布式电子商务平台，支持用户，订单，采购，物流，配送，财务等多个项目的协作，便于后续运营报表，分析，便于运维及监控。

架构演变

         基础框架剥离 -> 分库分表 -> 基础服务建设 -> 私有云建设 ->分布式操作系统

基础框架

         整个公司无论有多少项目，需要沉淀最基础的框架，里面一般包含核心的分库分表规则，统一的数据库操作类库，统一的通讯类，统一的日志类，统一的加密算法，统一的基础服务sdk，公用的一些工具类等等。该框架用于定义最基础的公司架构，设计，统一最基础的技术及项目架构规范，拦截及监控最基础的核心调用等。框架命名一般比较简单，如京东，可以定义为jdf；淘宝，可以定义为tbf。

分库分表

         分库分表为最常规的架构拆分方案。一般会从业务角度进行不同视角的拆分，如用户视角和商户视角。当然前提也需要业务方面或者其他技术力量的支持，不出现或者解决拆分后跨多个分库或者分表的表查询及查询结果合并问题。分库分表前也通过需要预估容量，预估性能。分库分表也经常会遇到全局id，或者分布式id自增且唯一的问题，这些都要预先在设计和架构层面要充分考虑。

*用户视角如图所示*

*商户视角如图所示*

基础服务

         基础服务是系统分布式的一个核心。它往往与操作系统基础组件相对应,只不过它是分布式的。如基础服务一般包含分布式存储，分布式缓存，分布式计算，分布式消息，分布式服务，分布式任务调度，分布式监控等。对应于操作系统的磁盘，内存，cpu，跨进行消息，进程，计划任务，系统监控等。

         公司的基础服务暂时包含几块: 分布式缓存,业务消息队列，任务调度，监控平台，服务中心，分布式锁服务，配置中心。

*基础服务如图所示*

**分布式缓存**（开源地址:<http://git.oschina.net/chejiangyi/XXF.BaseService.DistributedCache> 博文:<http://my.oschina.net/chejiangyi/blog/595038>）

         目前主要是解决核心几个页面的缓存问题，比如首页和列表页等，从而解决高频繁下频繁查询数据库的问题。一般来说缓存的内容越细越好，这样缓存的内容会比较多，对数据库的性能优化效果自然很佳。但是缓存越细，则关于缓存的清理工作就越细致，很容易代码编写过程中忘记清理缓存的情况，影响面和用户体验会很糟糕。

这种情况可能有两种方式解决，一种是架构上已经达到服务化和模块化层面，每个模块只处理自身相关的缓存。如用户服务，订单服务，商户服务，商品服务等，仅处理自身相关的缓存。那么缓存足够细，当然代码处理也能更加细致。另外一种是数据库或者其他层面的修改回调，批量清除相关的缓存；因为粒度很粗，但是可能会出现大量可用缓存被清理，造成部分雪崩效应。

所以我们经常会认为使用缓存很容易，但是用好缓存需要的是根据业务需求和许可设计缓存结构，尽量用好缓存，达到理想的性能；又或者我们只使用少量粗粒度的缓存，定义好缓存失效时间，部分代码清理部分缓存的方式来，这样能保证热点页面的较高性能; 但是这种情况下我们依然要注意缓存项不能太多，代码规范管理。

同时我们注意到业务的缓存会有一些特点，有些缓存具备高热点特性，有些缓存具备瞬间热点特性，有些缓存可以丢失，有些缓存尽量保证不丢失（否则可能造成雪崩效应）。故我们要根据实际业务不同，采用不同的存储介质。比如redis，memcache，ssdb等，应用场景都略微不同。而做为公司级的缓存中间件，应该适当的屏蔽这些存储特性，最好可以做到无缝配置，同时具备负载均衡，性能监控等等。

**业务消息队列**（开源地址: <http://git.oschina.net/chejiangyi/Dyd.BusinessMQ> 博文:<http://my.oschina.net/u/2379842/blog/515860>）

         主要是解决业务间的高可靠性消息传递，及功能的异步化处理。这种消息队列必须有以下几点：承载高并发，业务消息不允许丢失，业务消息必须能支撑超大量的堆积而稳定，支持消息的回溯。一般公司可能会考虑企业服务总线（esb），但是针对电子商务瞬间高并发和大量消息堆积的需求，可能不太合适，而且esb包含的东西很多，属于重量级的解决方案，更适合一般企业项目，如企业的内部管理系统。当然一些公司可能也会使用activemq，rabbitmq，kafka，metaq，tbnotify等等，具体的会根据使用场景和实际业务需求选择。比如一些内存的消息中间件，不支持持久化，不支持消息堆积，不支持消息回溯，其实不适合当前公司的业务场景，故而放弃或者部分场景使用。

**任务调度平台**（开源地址:<http://git.oschina.net/chejiangyi/Dyd.BaseService.TaskManager> 博文:<http://my.oschina.net/u/2379842/blog/484635>）

         主要是解决业务的后端任务挂载，隔离，定时调度，任务出错报警等。未来可以做到父子任务的关联，任务资源的自动分配和协调，任务的故障转移和均衡。那么网络爬虫，报表分析，弹性计算等资源型任务就可以适用了。

**统一监控平台**（开源地址：<http://git.oschina.net/chejiangyi/Dyd.BaseService.Monitor> 博文:<http://my.oschina.net/u/2379842/blog/510655>）

         每个公司对监控的需求其实都不一样，一般会根据业务不同，根据架构不同，根据基础服务不同，会不同程度的抓取和集成一些性能指标，业务日志，错误日志，耗时性能，流量等等至监控平台。市面上有很多大而全的监控平台，其实也都提供了sdk做二次开发，目的也在于此。

毕竟业务不同，服务器环境不同，架构基础设施不同，自然关注的性能点和指标参数也都不一样，故从长远发展，监控平台对整个分布式系统的稳定性是具备关键性作用。大型的分布式电子商务系统，监控平台本身就是大量系统性能分析师的分析思路，分析技巧的总结和沉淀。当然中小型的企业，可以直接使用第三方监控工具，但是性能分析往往是事后的，非及时性的；又或者第三方的监控工具很多，却没有有效的整合在一起，真正分析性能的时候却一片茫然。又或者大型项目单个操作涉及的系统或服务很多，需要拿到分布式的调用堆栈和调用链…. 种种这些都难免需要公司沉淀自己的监控平台。

**服务中心**（暂未开源）

主要是解决多个项目之间的同步调用，项目的公用api下沉，及远程调用服务的负载均衡，性能监控，预警等等。当前其本质上是服务的管理，公开，协调，运维等，满足业务soa的架构设计。特别是未来对业务细化拆分，模块化，同步解耦起关键作用,类似淘宝的HSF。

**分布式锁**（开源地址:<http://git.oschina.net/chejiangyi/XXF.BaseService.DistributedLock> 博文:<http://my.oschina.net/u/2379842/blog/511291>）

分布式锁服务的应用即便在大型业务项目中都不应该经常使用，特别是对性能要求较高的功能，不建议使用或谨慎使用。任何使用分布式锁的功能建议进行codereview和使用论证。理论上分布式锁常用于基础设施服务的分布式协调，但是一些业务对一致性要求较高，特别对瞬间并发导致相同业务同时执行的要求特别高，需要采用分布式锁，否则不采用。

**配置中心**（开源地址: <http://git.oschina.net/chejiangyi/Dyd.BaseService.ConfigManager> 博文:<http://my.oschina.net/u/2379842/blog/533604>）

主要是解决多项目之间的配置聚合及统一管理，同时具备配置的及时更新。若集成任务调度服务可以做到配置的负载均衡和配置的故障转移等。为什么要解决公司的项目配置？公司项目从页面前端，采购，物流，配送，财务，b2b等具有多个项目，多个项目各自同样具有多个服务，多个后台任务，这些程序都互相独立，且部分需要负载均衡（未来数量将达到上千个数量）但是又经常需要公用同一套配置体系。若每个程序都配置同一个配置信息，那么未来做某个配置信息的迁移或者更新，运维和开发人员根本不知道哪里需要更新。故配置的聚集管理在大型项目中是非常有必要的。而且基于配置中心也可以实现热切换，自动故障转移，软性负载等分布式的服务管理能力。

以上这些仅仅是目前公司根据业务发展需要使用的一些基础服务，当然很多公司也会根据自身的业务需求使用分布式存储，分布式搜索引擎等，也会根据自身对基础服务的可靠性要求，稳定性要求选择不同的开源基础服务框架进行改造，或者直接使用或者二次封装。从架构师的视角，针对业务发展，要谨慎选择，又要谨慎考虑是否需要重复造轮子。

文中介绍的相关基础服务可加入开源QQ群: **.net开源基础服务,238543768** 一起交流心得。

私有云、混合云建设

         很多创业公司初期，特别是电子商务类型的公司，可以会优先考虑第三方云计算平台来搭建整个平台和架构，自然更多的可能是基于成本，资源，人手方面考虑。但是当公司发展到一定程度，建立自己的机房可能是非常必要的选择。个人认为云计算的服务器（ecs）达到60-100台集群的时候，考虑搭建公司的机房已经非常有必要了，而当机房的物理机器达到20台的时候，可能也需要考虑放弃单纯的kvm，转而使用openstack及配合使用docker，container等容器技术会更加合适。

         公司对私有云的建设主要是偏重于物理机的资源合理利用，及私有云的有效，灵活管理，甚至必要的情况下，修改openstack的源代码，配合前端的流量和压力情况进行扩容和缩容，更加深层次的自动的负载均衡和自动的故障切换等等。

分布式操作系统

         分布式操作系统本身是一种概念思想的，本身未必具体的如何做的执行步骤。个人认为它更偏向于在云计算平台搭建后的资源更加有效整合，及平台在解决业务能力的稳定性和扩展能力；从架构师的视角看，也许更多的站在更高的层次全局的俯视整个平台架构，一个整体的电子商务的分布式操作系统和解决方案，而非仅仅是云计算平台。在这个阶段我们可以尝试修改openstack的源码和基础服务的源码，两者无缝结合起来，做到高流量时候自动的扩容及低流量时的自动缩容，做到资源的动态调配合。

*（本说明基于当前公司的实际情况的部分架构简要说明，未涉及工作流，搜索引擎，大数据挖掘等其他方面，仅作参考）*

# 【开源】.net 分布式架构之配置中心

[收藏](javascript:void(0);)

[车江毅](https://my.oschina.net/chejiangyi/home)

* 发表于 11个月前

* 阅读 1403

* 收藏 6

* 点赞 1

* [评论 2](https://my.oschina.net/chejiangyi/blog/533604#comment-list)

*摘要: .net 统一配置中心,用于所有项目统一的配置集中管理，简化运维和项目部署,具备高灵活性，高性能，高稳定性，高及时性。 同时具备配置项的负载均衡和故障转移,从而实现项目的软性的负载均衡和故障转移能力。*

**开源QQ群: .net 开源基础服务  238543768**

开源地址: <http://git.oschina.net/chejiangyi/Dyd.BaseService.ConfigManager>

**.net 统一配置中心**  
  
用于所有项目统一的配置集中管理，简化运维和项目部署,具备高灵活性，高性能，高稳定性，高及时性。  
同时具备配置项的负载均衡和故障转移,从而实现项目的软性的负载均衡和故障转移能力。  
  
1.高灵活性  
一个项目可以自由组合或者继承多个分类配置，多个分类的相同配置项可以被子分类配置项覆盖。  
（举例:项目A可以组合使用“基础数据库配置”，“基础监控配置”，“基础日志配置”，“项目A配置”;项目A配置可以定义自身的特殊配置和覆盖一些基础配置）  
  
2.高稳定性  
sdk（客户端）默认会在“本地”保存项目的所有配置“快照”，若统一配置中心异常，sdk将从上一次可用“快照”获取配置，并在配置中心恢复后，获取最新版本。  
  
3.高性能  
sdk（客户端）默认在本地内存中缓存“最新版本”的项目配置，sdk获取的配置都从内存版本中获取。  
  
4.高及时性  
  4.1 sdk（客户端）默认心跳时间内连接配置中心获取最新配置修改。  
  4.2 web端的配置修改，通过第三方消息中间件及时通知相关sdk至配置中心获取最新修改。  
  4.3 web端可以重启相关分类/相关项目的客户端，重新初始化所有的配置信息。  
  
5.配置负载均衡  
可以在web端手工配置某个配置的负载均衡（可以配置不同的使用权重命中不同的负载均衡配置）  
（若要自动，请编写任务挂载到“任务调度”中，根据业务情况，动态调整负载均衡权重。参考openapi接口）  
  
6.配置故障转移  
可以在web端手工配置某个配置的故障转移。  
（若要自动，请编写任务挂载到“任务调度”中，根据业务情况，动态监测当前配置的故障情况，然后选择使用备用故障配置。参考openapi接口）

**使用demo**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using XXF.BaseService.ConfigManager;//使用ConfigManager命名空间

namespace Dyd.BaseService.ConfigManager.Test

{

    /\*

     \* 配置中心使用demo

     \*/

    public class ConfigManagerDemo

    {

        /// <summary>

        /// 配置中心基本配置初始化

        /// </summary>

        public void InitConfig()

        {

            XXF.Common.XXFConfig.ProjectName = "projecttest";//当前项目名称,项目代码配置或者从App.config/web.config读取

            XXF.Common.XXFConfig.ConfigManagerConnectString = "server=192.168.17.111;Initial Catalog=dyd\_bs\_config;User ID=sa;Password=123456;";//配置中心管理数据库,项目代码配置或者从App.config/web.config读取

        }

        /// <summary>

        /// 使用demo

        /// </summary>

        public void UseDemo()

        {

            /\*

             \* 配置获取优先级

             \* 1.从本地app.config,web.config中优先获取

             \* 2.从配置中心获取次之。

             \*/

            string configkey = "configkey1";

            var value = ConfigManagerHelper.Get<string>(configkey);//获取配置值

        }

    }

}

**部分截图**

by 车江毅

# 【开源】.net 分布式架构之分布式缓存中间件

[收藏](javascript:void(0);)

[车江毅](https://my.oschina.net/chejiangyi/home)

* 发表于 10个月前

* 阅读 1057

* 收藏 4

* 点赞 0

* [评论 0](https://my.oschina.net/chejiangyi/blog/595038#comment-list)

*摘要: 放便实现缓存的分布式，集群，负载均衡，故障自动转移，并兼容多种缓存存储的.net分布式缓存中间件。*

开源git地址: <http://git.oschina.net/chejiangyi/XXF.BaseService.DistributedCache>

**分布式缓存中间件**  
  方便实现缓存的分布式，集群，负载均衡，故障自动转移，并兼容多种缓存存储的分布式缓存中间件。 用于解决[分布式架构](http://my.oschina.net/u/2379842/blog/521950)中的分布式缓存环节。

**特点**：  
 1. 代码少，便于扩展。  
 2. 兼容阿里云memcache，redis，ssdb。  
 3. 规范缓存使用接口，屏蔽底层缓存实现。  
 4. 通过配置连接字符串即可切换不同存储引擎，可以混合不同存储引擎组成缓存集群部署。（如部分redis，部分memcache）  
 5. 动态负载均衡，故障转移，线上无缝平行扩展和扩容，方便运维。

**不同存储介质**

        /// <summary>

        /// Redis

        /// 数据存内存,适合内存大小范围内大量缓存。（若是频繁失效的缓存数据，大量热点数据，建议使用redis）

        /// </summary>

        Redis,

        /// <summary>

        /// SSDB

        /// 数据热点存内存，大量数据存磁盘。（若是命中率较低，命中热点数据，大量冷数据，建议使用ssdb）

        /// </summary>

        SSDB,

        /// <summary>

        /// Memcached

        /// </summary>

        Memcached,

        /// <summary>

        /// SQLServer内存表

        /// </summary>

        SqlServer,

        /// <summary>

        /// 阿里云的缓存服务OCS

        /// </summary>

        AliyunMemcached,

**备注**：  
 1. 属于半研究性项目，已在线上阿里云memcache环境使用。

**未来发展**:  
 1. 分布式缓存中间件平台化，实现缓存监控，预警，性能报告等，性能数据收集至监控平台。  
 2. 扩展分布式缓存的其他特点。  
 3. 环形一致性hash对负载均衡和故障转移的支持。    
  
开源相关群: .net 开源基础服务 **238543768**  
(大家都有本职工作，也许不能及时响应和跟踪解决问题，请谅解。)  
  
by 车江毅

# .net erp（办公oa）开发平台架构之流程服务概要介绍

[收藏](javascript:void(0);)

[车江毅](https://my.oschina.net/chejiangyi/home)

* 发表于 10个月前

* 阅读 305

* 收藏 2

* 点赞 1

* [评论 2](https://my.oschina.net/chejiangyi/blog/599690#comment-list)

*摘要: .net erp（办公oa）开发平台架构之流程服务(流程引擎)概要介绍*

**背景**

搭建一个适合公司erp业务的开发平台。

**架构概要图**：

**流程引擎开发平台**：  
  包含流程引擎设计器，流程管理平台，流程引擎服务。目前只使用单个数据库进行管理。  
  **流程引擎设计器**

   采用silverlight进行开发，本质是对流程模型进行设计，并生成xml。包含:人工节点，自动节点，并行开始节点，并行结束节点，消息节点，文本节点。  
  示例模型定义图形：  
      
  示例模型定义xml：

<?xml version="1.0" encoding="gb2312"?>

<ProcessModel xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <BaseInfo>

    <Key>ddd</Key>

    <Title>新建模型</Title>

    <Description>发起人</Description>

    <CreateTime>0001-01-01T00:00:00</CreateTime>

    <Author>

      <StaffNo>34</StaffNo>

      <UserName>车江毅1</UserName>

    </Author>

    <Type>Common</Type>

    <Compilation>Debug</Compilation>

  </BaseInfo>

  <Variables>

    <Variable Name="myValue1" Value="今天天气好晴朗" Mode="Flow" />

    <Variable Name="myValue2" Value="v2" Mode="Flow" />

    <Variable Name="myValue3" Value="v3" Mode="Flow" />

    <Variable Name="myValue4" Value="true" Mode="Flow" />

  </Variables>

  <Nodes>

    <BaseNode xsi:type="StartNode" Key="start1" Text="开始" Location="236 10">

      <Routes>

        <Route To="auto1" Text="" Location="0 0">

          <RouteScripts />

        </Route>

      </Routes>

      <BeginScripts />

      <EndScripts />

    </BaseNode>

    <BaseNode xsi:type="AutoNode" Key="auto1" Text="系统判断" Location="236 95">

      <Routes>

        <Route To="man1" Text="线1" Location="0 0">

          <Code><![CDATA[myValue4=='true']]></Code>

          <RouteScripts />

        </Route>

        <Route To="人工2" Text="线" Location="0 0">

          <Code><![CDATA[myValue4=='false']]></Code>

          <RouteScripts />

        </Route>

      </Routes>

      <BeginScripts />

      <EndScripts />

      <Scripts>

        <Script>

          <Code><![CDATA[api.Forword('http://webservice.webxml.com.cn/webservices/ChinaTVprogramWebService.asmx','getTVstationDataSet','theAreaID:2')]]></Code>

        </Script>

        <Script>

          <Code><![CDATA[api.UpdateVariable('063550')]]></Code>

        </Script>

      </Scripts>

    </BaseNode>

    <BaseNode xsi:type="ManNode" Key="man1" Text="人工1" Location="167 205">

      <Routes>

        <Route To="并行开始1" Text="线" Location="0 0">

          <Code><![CDATA[]]></Code>

          <RouteScripts />

        </Route>

      </Routes>

      <BeginScripts />

      <EndScripts />

      <Participants>

        <Participant Mode="Code">

          <Code><![CDATA[api.Split('034','#')]]></Code>

        </Participant>

        <Participant Mode="Variable">

          <Code><![CDATA[myValue3]]></Code>

        </Participant>

      </Participants>

      <Solt OrderCount="2" Mode="Once" />

      <Decisions>

        <Decision Text="同意" />

        <Decision Text="不同意" />

      </Decisions>

    </BaseNode>

    <BaseNode xsi:type="EndNode" Key="结束" Text="结束" Location="238 732">

      <Routes />

      <BeginScripts />

      <EndScripts />

    </BaseNode>

    <BaseNode xsi:type="ManNode" Key="人工2" Text="人工2" Location="324 203">

      <Routes>

        <Route To="并行开始1" Text="线" Location="0 0">

          <Code><![CDATA[]]></Code>

          <RouteScripts />

        </Route>

      </Routes>

      <BeginScripts />

      <EndScripts />

      <Participants />

      <Decisions />

    </BaseNode>

    <BaseNode xsi:type="ParallelBeginNode" Key="并行开始1" Text="并行签入" Location="237.047607421875 312">

      <Routes>

        <Route To="人工3" Text="线" Location="0 0">

          <Code><![CDATA[]]></Code>

          <RouteScripts />

        </Route>

        <Route To="人工4" Text="线" Location="0 0">

          <Code><![CDATA[]]></Code>

          <RouteScripts />

        </Route>

      </Routes>

      <BeginScripts />

      <EndScripts />

    </BaseNode>

    <BaseNode xsi:type="ParallelFinishNode" Key="并行结束1" Text="并行结束1" Location="240 571">

      <Routes>

        <Route To="消息1" Text="线" Location="0 0">

          <Code><![CDATA[]]></Code>

          <RouteScripts />

        </Route>

      </Routes>

      <BeginScripts />

      <EndScripts />

    </BaseNode>

    <BaseNode xsi:type="ManNode" Key="人工3" Text="人工3" Location="95 378">

      <Routes>

        <Route To="自动1" Text="线" Location="0 0">

          <Code><![CDATA[]]></Code>

          <RouteScripts />

        </Route>

      </Routes>

      <BeginScripts />

      <EndScripts />

      <Participants />

      <Decisions />

    </BaseNode>

    <BaseNode xsi:type="ManNode" Key="人工4" Text="人工4" Location="392 379">

      <Routes>

        <Route To="自动2" Text="线" Location="0 0">

          <Code><![CDATA[]]></Code>

          <RouteScripts />

        </Route>

      </Routes>

      <BeginScripts />

      <EndScripts />

      <Participants />

      <Decisions />

    </BaseNode>

    <BaseNode xsi:type="AutoNode" Key="自动1" Text="自动1" Location="98 483">

      <Routes>

        <Route To="并行结束1" Text="线" Location="0 0">

          <Code><![CDATA[]]></Code>

          <RouteScripts />

        </Route>

      </Routes>

      <BeginScripts />

      <EndScripts />

      <Scripts />

    </BaseNode>

    <BaseNode xsi:type="AutoNode" Key="自动2" Text="自动2" Location="394.714294433594 474.952362060547">

      <Routes>

        <Route To="并行结束1" Text="线" Location="0 0">

          <Code><![CDATA[]]></Code>

          <RouteScripts />

        </Route>

      </Routes>

      <BeginScripts />

      <EndScripts />

      <Scripts />

    </BaseNode>

    <BaseNode xsi:type="MessageNode" Key="消息1" Text="消息1" Location="239 656">

      <Routes>

        <Route To="结束" Text="线" Location="0 0">

          <Code><![CDATA[]]></Code>

          <RouteScripts />

        </Route>

      </Routes>

      <BeginScripts />

      <EndScripts />

      <Participants />

      <MessageVariables />

      <TemplateKey>0</TemplateKey>

      <TemplateName />

    </BaseNode>

  </Nodes>

</ProcessModel>

  常规界面如下：  
   

**流程管理平台**  
  包含：流程模型管理，流程管理，流程任务管理，流程操作记录，流程服务接口日志查询，流程异常查询，流程流转调试日志查询，开发者用户管理等。  
  可以管理流程模型历史版本及版本切换，流程异常时候的流程节点切换，流程任务的转交等。  
 界面如下：

**流程服务**  
  包含：流程接口和流程任务接口。

    /// <summary>

    /// 业务流程服务接口

    /// </summary>

    public interface IProcessService

    {

        /// <summary>

        /// 获取业务流程

        /// </summary>

        /// <param name="guid">业务流程ID</param>

        ProcessInfo GetProcessById(Guid guid);

        /// <summary>

        /// 批量获取业务流程

        /// </summary>

        /// <param name="guids">多个业务流程ID</param>

        ProcessListInfo GetProcessesByIds(List<Guid> guids);

        /// <summary>

        /// 根据流程发布者，获取业务流程列表

        /// </summary>

        /// <param name="user">用户的信息 参看UserInfo对象注释</param>

        /// <param name="modelkeys">模型key  可选（多个）</param>

        /// <param name="keyword">关键词(标题) 可选</param>

        /// <param name="pageIndex"></param>

        /// <param name="pageSize"></param>

        /// <returns></returns>

        ProcessListInfo GetProcessListByOriginator(UserInfo user, List<string> modelkeys, string keyword, int pageIndex, int pageSize);

        /// <summary>

        /// 根据流程参与者，获取业务流程列表

        /// </summary>

        /// <param name="user">用户的信息 参看UserInfo对象注释</param>

        /// <param name="modelkeys">模型key 可选（多个）</param>

        /// <param name="keyword">关键词(标题) 可选</param>

        /// <param name="nodekey">任务所在节点key 可选</param>

        /// <param name="state">任务完成状态</param>

        /// <param name="pageIndex"></param>

        /// <param name="pageSize"></param>

        /// <returns></returns>

        ProcessListInfo GetProcessListByParticipant(UserInfo user, List<string> modelkeys, string keyword, string nodekey, EnumTaskStateQuery state, int pageIndex, int pageSize);

        /// <summary>

        /// 根据模型keys, 获取业务流程列表

        /// </summary>

        /// <param name="modelkeys">模型关键词 （多个）</param>

        /// <param name="keyword">关键词(标题) 可选</param>

        /// <param name="pageIndex"></param>

        /// <param name="pageSize"></param>

        /// <returns></returns>

        ProcessListInfo GetProcessListByModel(List<string> modelkeys, string keyword, int pageIndex, int pageSize);

        /// <summary>

        /// 创建业务流程

        /// </summary>

        /// <param name="modelkey">业务流程对应的模型key</param>

        /// <param name="title">业务流程标题</param>

        /// <param name="description">业务流程描述</param>

        /// <param name="variables">业务流程使用变量 可选</param>

        /// <param name="user">业务流程发起人 参看UserInfo对象注释</param>

        /// <param name="operationArgs">操作记录 可选</param>

        /// <returns></returns>

        ProcessInfo CreateProcess(string modelkey, string title, string description, List<Variable> variables, UserInfo user, OperationRecordInfoArgs operationArgs);

        /// <summary>

        /// 更新业务流程

        /// </summary>

        /// <param name="guid">业务流程的ID</param>

        /// <param name="title">业务流程的标题</param>

        /// <param name="description">业务流程的描述</param>

        /// <param name="variables">业务员流程的变量 可选</param>

        void UpdateProcess(Guid guid, string title, string description, List<Variable> variables);

        /// <summary>

        /// 结束业务流程

        /// </summary>

        /// <param name="guid">业务流程的ID</param>

        void FinishProcess(Guid guid, UserInfo user, OperationRecordInfoArgs operationArgs);

        /// <summary>

        /// 创建流程操作记录

        /// </summary>

        /// <param name="user">操作用户不能为空</param>

        /// <param name="operationArgs">操作记录不能为空</param>

        void CreateProcessOperationRecord(Guid guid, UserInfo user, OperationRecordInfoArgs operationArgs);

    }

/// <summary>

    /// 任务服务接口

    /// </summary>

    public interface ITaskService

    {

        /// <summary>

        /// 根据任务id，获取任务

        /// </summary>

        /// <param name="id">任务id</param>

        /// <returns></returns>

        TaskInfo GetTaskByID(string id);

        /// <summary>

        /// 根据任务id，批量获取任务

        /// </summary>

        /// <param name="ids">多个任务id</param>

        /// <returns></returns>

        TaskListInfo GetTasksByIDs(List<string> ids);

        /// <summary>

        /// 根据业务流程id，获取任务信息列表

        /// </summary>

        /// <param name="state">任务状态</param>

        /// <param name="processId">业务流程id</param>

        /// <returns></returns>

        TaskListInfo GetTaskListByProcessID(EnumTaskStateQuery state, Guid processId);

        /// <summary>

        /// 根据用户，获取任务信息列表

        /// </summary>

        /// <param name="user">用户的信息 参看UserInfo对象注释</param>

        /// <param name="state">任务状态</param>

        /// <param name="pageIndex"></param>

        /// <param name="pageSize"></param>

        /// <returns></returns>

        TaskListInfo GetTaskListByUser(UserInfo user, EnumTaskStateQuery state, int pageIndex, int pageSize);

        /// <summary>

        /// 根据用户和流程，获取任务信息列表

        /// </summary>

        /// <param name="user">用户的信息 参看UserInfo对象注释</param>

        /// <param name="processId">业务流程id</param>

        /// <param name="nodekey">当前活动节点key 可选</param>

        /// <param name="state">任务状态</param>

        /// <param name="pageIndex"></param>

        /// <param name="pageSize"></param>

        /// <returns></returns>

        TaskListInfo GetTaskListByUserAndProcess(UserInfo user, Guid processId, string nodekey,EnumTaskStateQuery state, int pageIndex, int pageSize);

        /// <summary>

        /// 根据业务流程模型key，获取任务信息列表

        /// </summary>

        /// <param name="user">用户的信息 参看UserInfo对象注释</param>

        /// <param name="modelkeys">业务模型key 可选（多个）</param>

        /// <param name="nodekey">当前活动节点key 可选</param>

        /// <param name="state">任务状态</param>

        /// <param name="pageIndex"></param>

        /// <param name="pageSize"></param>

        /// <returns></returns>

        TaskListInfo GetTaskListByUserAndModel(UserInfo user, List<string> modelkeys, string nodekey, EnumTaskStateQuery state, int pageIndex, int pageSize);

        /// <summary>

        /// 执行任务

        /// </summary>

        /// <param name="id">任务id</param>

        /// <param name="action">任务的动作 例如：“同意”“不同意”</param>

        /// <param name="variables">流程参数 可选</param>

        /// <param name="operationArgs">操作信息 必填</param>

        TaskInfo ExecuteTask(string id, string action, List<Variable> variables, OperationRecordInfoArgs operationArgs);

        /// <summary>

        /// 转交任务

        /// </summary>

        /// <param name="id">任务id</param>

        /// <param name="toUser">任务转交人信息 参看UserInfo对象注释</param>

        /// <param name="operationArgs">操作信息 传null则无转交操作记录</param>

        TaskInfo TransferTaskWithOperationRecordInfoArgs(string id, UserInfo toUser, OperationRecordInfoArgs operationArgs);

**分布式架构概想**：  
  分布式拆库方案：流程维度和用户维度，流程维度数据库按照流程定义模型的唯一标识hash，来拆分到不同的数据库。所有流程创建和流程任务的创建都在不同的数据库中。用户维度数据库按照用户标识hash，来拆分到不同的数据库。  
  拆库优点：数据被拆分，不影响同一个业务流程流转，性能会得到提升。  
  拆库缺点：采用异步消息通知，做用户代办任务和完成任务冗余及同步。采用BI工具或者hadoop等进行数据报表等分析。开发，维护复杂度等都会提升。  
  结论：如果不是做大型产品或服务，不会进行分布式架构。

*（此文只做阶段性的总结，也许对同样做流程引擎的人有些启发，也欢迎交流。分布式相关架构可以参考本人其他文章）*

# .net erp（办公oa）开发平台架构概要说明之表单设计器

[收藏](javascript:void(0);)

[车江毅](https://my.oschina.net/chejiangyi/home)

* 发表于 10个月前

* 阅读 427

* 收藏 4

* 点赞 1

* [评论 0](https://my.oschina.net/chejiangyi/blog/600645#comment-list)

*摘要: .net erp（办公oa）开发平台架构概要说明之表单设计器介绍*

**背景**：搭建一个适合公司erp业务的开发平台。

**架构概要图**：

**表单设计开发部署示例图**

**表单设计开发部署示例说明**  
1）每个开发人员可以自己部署表单设计至本地一份（当然也可以共用一套开发环境，但是如果开发交叉功能也比较容易冲突和调试冲突，特别是服务经常调试产生冲突）。完成某个功能开发以后，通过项目一键发布至测试环境进行集成测试。在集成测试ok后，一键发布至线上开发环境。  
2）正式线上部署可以按照项目为单位建立负载均衡，按照项目为单位整体建立集群。单个web站点包含表单文件服务和表单代码服务。表单文件服务为一些css，js，html，图片等的静态文件。表单代码服务为一些动态C#代码服务，随时编译，随时发布，以http方式提供服务。表单文件可以通过js调用这些http服务。

**表单项目管理功能**  
图：

说明：  
表单设计按照项目的方式管理，表单文件服务和表单代码服务及各自相应的编辑及文件。所以建立表单设计前，应该要先建立项目。  
项目名称是整个项目发布，开发的唯一约定名称，不得重复。  
若有公用的项目，则建立公用的项目，里面可包含公用的js，公用的图片，公用的css等。这个也是所有表单项目可公用的基础前端框架。  
发布和撤销服务按钮：用于发布当前的c#后端代码成服务接口。若当前项目设置了负载均衡，相应的负载均衡节点会在2s-5s内自动更新服务。  
发布上线按钮：用于发布当前项目至另外一个项目环境。比如测试或线上环境。目前至支持全量发布，未来可优化。  
清除缓存按钮：清理表单文件服务的静态文件缓存。因为表单文件都是会在单个网站存储静态文件缓存，从而降低数据库的压力和静态文件负载均衡实现。理论上静态文件更新，本身相应项目的负载均衡节点也会在2s-5s内更新相应的缓存文件。

**表单文件管理**  
列表图

前端详情图

后端详情图

全屏编辑示例：

表单文件包含前端文件（css，js，html，图片等静态文件）和后端文件（.cs 的c#后端代码文件）这两种文件，相应的不同类型文件会形成相应的不同服务。  
  
***前端文件相关介绍***  
前端文件包含（css，js，html，图片等静态文件），其中可以通过上传附件的方式如（jquery框架.js文件）,也可通过编辑文本的方式提交。会根据不同的后缀进行解析输出，同时支持跨域访问。  
历史版本：默认根据文件名保留历史版本，故也可恢复使用历史版本。  
内部方式按钮：不使用站点缓存文件访问，直接访问数据库输出文件。  
外部方式按钮：会使用站点缓存文件访问输出文件。  
调试预览按钮：可以打开新的页面进行当前编辑效果的实时调试，一键刷新，所见即所得。  
（编辑器控件，参考http://runjs.cn/code）  
  
***后端文件相关介绍***  
后端文件包含.cs c#代码文件；c#后端文件分为接口文件和公用代码文件两种类型。  
接口文件：会自动公开该文件的类的方法为接口，支持http访问。js即可调用。  
公用代码文件：该类默认为static静态类及静态方法。可被其他类访问。  
第三方程序集的支持：第三方程序集放到站点的BuildsAssemblys文件下面，即可被其他类库所调用。（目前不支持第三方程序集的自动发布，正式发布要手工拷贝到线上）  
快速编译：根据本地的cs文件缓存，快速编译。  
重新编译：重新下载当前项目所有的cs文件至本地，重新编译。  
提交并发布：先重新编译->将当前内容保存到数据库(非草稿保存)->发布当前服务到所在站点（不会发布到负载均衡节点）。  
代码调试：目前仅支持接口通过模拟http的方式调试，方便接口调试。  
  
***其他介绍***：  
保存草稿：编辑器默认在进行编辑后每10s保存一次草稿到数据库。  
  
***备注说明***：  
个人更倾向于绝对定位形式的表单设计器，类似ccflow。但是因为公司的原因，决定使用常规页面设计的方式。而页面展现形式因为多变，普通的页面表单设计器不能满足。  
因为人手，成本，不开源等原因，目前仅为粗糙之作，未做很细性能考虑，仅考虑架构扩展和平行性能扩展。

by 车江毅

# .net Elasticsearch 学习入门笔记

[收藏](javascript:void(0);)

[车江毅](https://my.oschina.net/chejiangyi/home)

* 发表于 9个月前

* 阅读 304

* 收藏 1

* 点赞 1

* [评论 0](https://my.oschina.net/chejiangyi/blog/607232#comment-list)

*摘要: .net Elasticsearch （es） 学习入门笔记及简要总结。*

**一. es安装相关**  
*1.elasticsearch安装*  
  运行http://localhost:9200/  
*2.head插件*  
*3.bigdesk插件安装*  
(安装细节百度:windows elasticsearch 安装，有详细内容)

**二. es插件相关**  
http://www.searchtech.pro/elasticsearch-plugins (es 插件大全)  
https://github.com/medcl/elasticsearch-analysis-ik (ik 项目)  
https://github.com/lmenezes/elasticsearch-kopf (ES的插件  监控 节点状态 ， 也可以调试你的ES查询)  
https://github.com/medcl/elasticsearch-rtf/tree/master (2.1.1 + 1.6ik  还有拼音  和其他分词器 集成好了)  
https://github.com/lmenezes/elasticsearch-kopf (类似head的插件)  
https://www.elastic.co/downloads/marvel （监控ES健康状态）  
konf插件 (据说能看集群负载)

**三.es C# 客户端示例**

*1. 包下载elasticsearch.net,nest 组件。*  
   nest组件会依赖下载elasticsearch组件。  
   使用文档：http://nest.azurewebsites.net/  
   备注：.net 使用es的相对少一些，es版本更新也很快，很多使用也都是靠自己揣摩。多交流，多分享。  
*2. 创建连接客户端*  
     public ElasticClient GetClient()  
        {  
            var node = new Uri("http://192.168.17.54:9200");  
  
            var settings = new ConnectionSettings(  
                node,  
                defaultIndex: "my-application"  
  
            );  
            return new ElasticClient(settings);  
        }  
*3. 创建索引模型（索引结构）*  
[ElasticType(IdProperty = "Id", Name = "Person")]  
        public class Person  
        {  
           [ElasticProperty(Name = "Id", Type = FieldType.String, Index = FieldIndexOption.NotAnalyzed)]  
            public string Id { get; set; }  
            public string Firstname { get; set; }  
            public string Lastname { get; set; }  
            public string[] Chains { get; set; }  
           [ElasticProperty(Name = "content", Type = FieldType.String, Index = FieldIndexOption.Analyzed, Analyzer = "ik\_max\_word")]  
            public string Content { get; set; }  
        }  
     备注：其他人提供的model示例，更详细些，可下载附件。  
*4. 索引内容（创建索引）*  
  private void btnIndex\_Click(object sender, EventArgs e)  
        {  
            //var client = new ElasticsearchClient();  
            ////index a document under /myindex/mytype/1  
            //var indexResponse = client.Index("myindex", "mytype", "1", new { Hello = "World" });  
            var client2 = GetClient();  
  
            //client2.CreateIndex("test");  
            //client2.Map<Person>(c => c.MapFromAttributes());  
  
            IEnumerable<Person> persons = new List<Person>  
            {  
                new Person()  
                {  
                    Id = "4",  
                    Firstname = "aaa",//Boterhuis-040  
                    Lastname = "Gusto-040",  
                    Chains = new string[]{ "a","b","c" },  
                },  
                new Person()  
                {  
                    Id = "5",  
                    Firstname = "sales@historichousehotels.com",  
                    Lastname = "t Boterhuis 1",  
                    Chains = new string[]{ "a","b","c" },  
                },  
                new Person()  
                {  
                    Id = "6",  
                    Firstname = "Aberdeen #110",  
                    Lastname = "sales@historichousehotels.com",  
                    Chains = new string[]{ "a","b","c" },  
                },  
                new Person()  
                {  
                    Id = "7",  
                    Firstname = "Aberdeen #110",  
                    Lastname = "t Boterhuis 2",  
                    Chains = new string[]{ "a","b","c" },  
                },  
                 new Person()  
                {  
                    Id = "8",  
                    Firstname = "Aberdeen #110",  
                    Lastname = "t Boterhuis 2",  
                    Chains = new string[]{ "a","b","c" },  
                },  
            };  
            //foreach(var p in persons)  
            client2.IndexMany<Person>(persons,"test");  
        }  
*5. 简单搜索示例*  
  var client = GetClient();  
            var rs = client.Search<Person>(s => s.Index("test").QueryString(this.textBox1.Text));  
            this.richTextBox1.Text = JsonConvert.SerializeObject(rs.Documents);  
*6. 索引更新*  
 private void btnUpdate\_Click(object sender, EventArgs e)  
        {  
            var client2 = GetClient();  
  
            client2.Update<Person, object>(u => u  
             .Index("test")  
            .Id(4)  
            .Doc(new { Id="4", Firstname = "United States" })  
            .RetryOnConflict(3)  
            .Refresh()  
           );  
  
            //var u1 =  new Person()  
            //    {  
            //        Id = "4",  
            //        Firstname = "Boterhuis-040",  
            //        Lastname = "Gusto-040",  
            //        Chains = new string[]{ "a","b","c" },  
            //    };  
            //var u2 = new Person()  
            //    {  
            //        Id = "4",  
            //        Firstname = "United States",  
            //        Lastname = "Gusto-040",  
            //        Chains = new string[] { "a", "b", "c" },  
            //    };  
            //client2.Update<Person,Person>(u1,u2).  
        }  
  
7. 索引删除  
 private void btnDelete\_Click(object sender, EventArgs e)  
        {  
            var client2 = GetClient();  
            client2.DeleteIndex("test");  
            client2.DeleteIndex("my-application");  
        }  
*8. 总结*  
以上示例代码，简单的应用已经足够用。其他的就是高亮和分组。可以看文档。  
  
**四. es 集群**  
 es 默认是一个集群，相对solr云来说配置更简单，搭建更方便些。但是更多还是要根据业务进行自己的集群设计还是好费很多时间，很多精力。（上手容易，用好难）  
  
**五. es 与solr 对比**  
个人目前了解的：  
原来solr资料比较多，现在貌似es的资料更多一点。solr是官方英文pdf，es也是英文的。  
原来solr还有中文书籍，现在貌似没有了。es 目前还有几本书籍，但是讲的es版本略有老旧。  
solr上手相对es略微难些。  
（n年前，我用的是solr，那时候还没有solrcloud；es还没有出来，那时候solr资料反而多。个人也实现了自己的solr集群方案。其他的功能上的对比，还是百度，不重复。）  
  
**六. 如果es客户端调试请求**  
建议下载HTTPAnalyzer之类的tcp拦截工具。这样可以拦截验证sdk出来的请求连接，对比资料和书籍看下哪些参数写错了，对调试很有帮助。  
  
**七. es 附录**  
es术语介绍：  
cluster：  
  
代表一个集群，集群中有多个节点，其中有一个为主节点。这个主节点是可以通过选举产生的。注意，主从节点是对于集群内部来说的。es的一个概念就是去中心化，字面上理解就是无中心节点，这是对于集群外部来说的，因为从外部来看es集群，在逻辑上是个整体，你与任何一个节点的通信和与整个es集群通信是等价的。  
shards  
  
代表索引分片。es可以把一个完整的索引分成多个分片，这样的好处是可以把一个大的索引拆分成多个，分布到不同的节点上。构成分布式搜索。分片的数量只能在索引创建前指定，并且索引创建后不能更改。  
replicas  
  
代表索引副本，es可以设置多个索引的副本。副本的作用，一是提高系统的容错性，当某个节点的某个分片损坏或丢失时可以从副本中恢复，二是提高es的查询效率，es会自动对搜索请求进行负载均衡。  
recovery  
  
代表数据恢复或叫数据重新分布，es在有节点加入或退出时会根据机器的负载对索引分片进行重新分配，挂掉的节点重新启动时也会进行数据恢复。  
river  
  
代表es的一个数据源，也是其他存储方式（如：数据库）同步数据到es的一个方法。它是以插件方式存在的一个es服务，通过读取river中的数据并把它索引到es中，官方的river有couchDB的，RabbitMQ的，Twitter的，Wikipedia的。  
gateway  
  
代表es索引快照的存储方式。es默认是先把索引存放到内存中，当内存满了时再持久化到本地硬盘。gateway对索引快照进行存储，当这个es集群关闭再重新启动时，就会从gateway中读取索引备份数据。es支持多种类型的gateway，有本地文件系统（默认），分布式文件系统，Hadoop的HDFS和amazon的s3云存储服务。  
discovery.zen  
  
代表es的自动发现节点机制。es是一个基于p2p的系统，它先通过广播寻找存在的节点，再通过多播协议来进行节点之间的通信，同时也支持点对点的交互。  
Transport  
  
代表es内部节点或集群与客户端的交互方式。默认内部是使用tcp协议进行交互，同时它支持http协议（json格式）、thrift、servlet、memcached、zeroMQ等的传输协议（通过插件方式集成）。

# .Net 大型分布式基础服务架构横向演变概述

[收藏](javascript:void(0);)

[车江毅](https://my.oschina.net/chejiangyi/home)

* 发表于 8个月前

* 阅读 1736

* 收藏 37

* 点赞 2

* [评论 7](https://my.oschina.net/chejiangyi/blog/624765#comment-list)

*摘要: .Net 大型分布式基础服务架构横向演变概述,包含分布式任务调度，分布式配置中心，分布式消息队列，分布式缓存平台，分布式服务中心，分布式数据库中间件平台等基础服务的演变说明。*

一. 业务背景

     构建具备高可用，高扩展性，高性能，能承载高并发，大流量的分布式电子商务平台，支持用户，订单，采购，物流，配送，财务等多个项目的协作，便于后续运营报表，分析，便于运维及监控。

二. 基础服务架构说明

*参考“大型电子商务架构说明”.doc*

*（或http://my.oschina.net/chejiangyi/blog/521950）*

三. 基础服务架构横向演进架构图

四. 基础服务横向演进架构概述

1. **分布式任务调度平台演进**

*（开源地址:*[*http://git.oschina.net/chejiangyi/Dyd.BaseService.TaskManager*](http://git.oschina.net/chejiangyi/Dyd.BaseService.TaskManager)*博文:*[*http://my.oschina.net/u/2379842/blog/484635*](http://my.oschina.net/u/2379842/blog/484635)*）*

分布式任务调度平台演进方向主要有两个不同方向：分布式任务资源调度平台和分布式任务流调度平台，这个两个方向最终会形成两个不同的系统平台。

***1）分布式任务资源调度平台***

    所有操作系统都将安装java/.net任务管理器服务（类似docker的管理节点），每个任务管理器里面可以动态运行多个资源任务，所有java/.net服务或者任务都将视为基础的资源任务(类似docker的容器概念)。此平台用于整个公司业务基础资源管理（类似docker的管理系统）。能够实现服务/任务的，负载均衡（网络，cpu，内存，流量），故障转移，是整个弹性基础服务管理平台的基础。

      使用场景：为了实现基础服务的弹性调度和管理。未来所有业务服务或者后台任务都以“任务”的形式存在，遇到高并发，大流量，硬件压力，自动伸缩（自动扩展任务负载均衡到其他节点）来扩展容量和抗负载能力（在分布式弹性基础服务管理平台中配置管理）。

***2）分布式任务流调度平台***

    用于创建流程式任务，用于多任务之间的协作和运行。（类似创建办公流程一样的多协作形式的任务，并根据任务的执行结果进行流程的流转。也可以入hadoop一样分布式任务运行）

    使用场景：可以以此基础实现类似风控系统（单个订单进来，多个任务进行风险判断的规则引擎，每个规则都是一个任务），大型的数据统计和抽取（可以实现map reduce之类的），分布式爬虫任务（运行一个流程，创建多个子爬虫任务不断运行）。

**2. 分布式配置中心平台演进**

*（开源地址:*[*http://git.oschina.net/chejiangyi/Dyd.BaseService.ConfigManager*](http://git.oschina.net/chejiangyi/Dyd.BaseService.ConfigManager)*博文:*[*http://my.oschina.net/u/2379842/blog/533604*](http://my.oschina.net/u/2379842/blog/533604)*）*

分布式配置中心的演进方向主要会集成两块平台：分布式集群高可用管理平台和分布式服务降级保障平台。当然基础的分布式配置中心的功能将会保留，这两个平台的功能前期会集成入配置中心（后期发展到一定复杂度会从配置中心单独剥离出来，但是又依赖基础的配置中心）。

***1）分布式集群高可用管理平台***

    这是基于配置中心（也支持轮询回调）的软性负载均衡，故障检测预警，故障转移实现的统一管理和检测平台，与keepalive之类的软件有些类似，会支持数据库，网站，第三方软件等。

***2）分布式服务降级保障平台***

    这是基于配置中心的服务、功能降级保障平台，前期会进行降级配置的统一管理和人工手动降级（后期一般会根据服务的cpu，内存，流量，相应时间等状况，自动进行降级，这时可以考虑单独扩展成一个平台）

**3. 分布式监控平台演进**

*（开源地址：*[*http://git.oschina.net/chejiangyi/Dyd.BaseService.Monitor*](http://git.oschina.net/chejiangyi/Dyd.BaseService.Monitor)*博文:*[*http://my.oschina.net/u/2379842/blog/510655*](http://my.oschina.net/u/2379842/blog/510655)*）*

分布式监控平台演进方向主要是这几块的功能扩展：分布式数据库监控平台，分布式缓存监控平台，分布式服务器集群监控预警平台，分布式业务监控平台，分布式日志监控分析预警平台等。这几块的功能扩展，全部以插件架构形式集成入监控平台。包括以后根据基础服务和平台演进的要求，越来越多的监控插件会集成入监控平台，而非单纯依赖第三方监控（任何一个高要求的大型网站，必须建立自身的监控体系）。

***1）分布式数据库监控平台***

    收集各种dba常规的sqlserver或mysql数据库的信息汇总，用于分析问题语句，及问题语句自动预警，及一些自动化的索引建议，同时提供cpu，内存，io，sql阻塞等情况的预警。（特别是大量数据库分库分表的情况下，需要集中优化与预警，及sql性能下降的提醒等）

***2）分布式缓存监控平台***

    可以是单纯的某种分布式缓存的监控，如redis，memcache，ssdb等。分布式缓存中间件平台会在自身平台中有监控数据，前期不集成到这里。当然开源社区也会有很多第三方的相关监控，但是如果想实现自身的一些特殊要求，比如统一的多维度预警就难以实现，特殊分析等，前期具体看情况而定，后期必定自研一套。

***3）分布式服务器集群监控平台***

    用于linux,windows的集群监控，根据配置支持多种操作系统指标的监控支持。操作系统级别的监控重要性就不必说了，也有很多第三方的相关监控工具，具体的也要看情况而定，但是涉及到预警这块还是必须自研。

***4）分布式业务监控平台***

    用于业务级别的监控，如api,业务sql，一些业务方法调用频率耗时，及类似百度站长工具的一些行为分析（这块做的东西就很深入了，需要大数据分析）等。

***5）分布式日志监控分析预警平台***

    用于汇集整个业务线，基础服务平台错误日志分析及分等级预警，关键业务日志打印分析等，这块是监控平台前期必须自研和统一的。

**4. 分布式消息队列演进**

*（开源地址:*[*http://git.oschina.net/chejiangyi/Dyd.BusinessMQ*](http://git.oschina.net/chejiangyi/Dyd.BusinessMQ)*博文:*[*http://my.oschina.net/u/2379842/blog/515860*](http://my.oschina.net/u/2379842/blog/515860)*）*

分布式消息队列的演进主要是未来满足公司对不同类型的消息队列类型的稳定性及性能要求等（目前消息队列相对成熟），主要有几方面扩展：

   1） 支持push方式的消息推送。

   2） 插件化剥离底层消息存储的单一依赖，支持多种存储扩展（内存，文件，数据库）等。

   3） 外围接口兼容actviemq，rabbitmq等多种消息存储及形式。

   4） 支持消息的事务化消费（多方业务订阅消费，一方消费失败则所有消费回滚，否则消息消费出错）

   5） 消息的服务化（broker），支持http，thrift协议等，便于跨语言使用。

   6)  弹性消费能力和弹性扩容等支持。

**5. 分布式缓存平台演进**

*（开源地址:*[*http://git.oschina.net/chejiangyi/XXF.BaseService.DistributedCache*](http://git.oschina.net/chejiangyi/XXF.BaseService.DistributedCache)*博文:*[*http://my.oschina.net/chejiangyi/blog/595038*](http://my.oschina.net/chejiangyi/blog/595038)*）*

目前分布式缓存做的很简单，只是简单的一个sdk代理机制。未来分布式缓存平台演进方向主要有以下几点：

   1） 以redis协议为模板，支持多种缓存存储介质。

   2） 支持一致性（环形）等多种hash分片方式。

   3） 强大的监控及管理平台。

   4） 支持缓存的桶迁移，支持缓存的主备（从），故障转移，负载均衡等。

   5） 缓存的服务化（proxy），支持http，thrift协议等，便于跨语言使用。

6)  弹性缓存扩容等支持。

**6. 分布式服务中心平台演进**

*（暂未开源,开源计划中）*

分布式服务中心平台要保持轻量级和高性能，未来演进方向应该包含以下几点：

   1）支持多种服务通信协议（thrift，自定义协议）。

   2）支持tcp和http。

   3）支持服务负载均衡和故障转移。

   4）强大的监控管理平台（耗时，连接数，cpu等性能，调用链，熔断机制，服务文档等）

5）弹性服务抗压支持。

**7. 分布式web版本发布管理平台**

   分布式web版本发布管理平台主要包含以下两块内容：

   1.用于公司项目web的统一版本控制，负载均衡节点统一发布和回滚。

   2.未来公司手机h5页面版本控制和版本管理。

**8. 分布式数据库管理平台**

   分布式数据库管理平台主要包含两块内容：分布式数据库中间件平台，分布式数据库运维平台。

***1）分布式数据库中间件平台***

主要集成数据库中间件功能，如分库分表sharding机制，sql拦截监控，sql耗时分析，优化建议等，类似tddl及mycat，细节不再赘述。

***2）分布式数据库运维平台***

分布式数据库集群的监控及运维管理功能，分布式数据库迁移功能，数据库运维工具，脚本及运维日志等。

**9. 分布式弹性基础服务管理平台**

   分布式弹性基础服务管理平台除了包含自身平台外，还包含分布式高并发作战平台。

***1）分布式高并发作战平台***

    用于抢购，秒杀时的一个作战平台，该平台将所有基础服务的外围核心监控，服务升降级配置，手工扩容等相关配置项目快捷的，整合一起为多级降级方案。

***2）分布式弹性基础服务管理平台***

    用于结合分布式任务资源管理平台，分布式监控，分布式消息队列，分布式服务中心，分布式配置中心等所有基础平台的可控制基础服务及业务服务/任务自动弹性伸缩，故障恢复的配置，管理，监控平台。

    使用场景：

    1）某个业务服务突然间流量超过阀值，通过分布式任务资源管理平台，将服务扩展到1/n台负载均衡服务，当流量低于某个阀值时自动回收服务。

    2）当某个业务的消息量大量堆积，通过分布式任务资源管理平台，增加业务消息消费任务负载均衡，当消息堆积量低于某个阀值时回收任务。

    3）当某个后台任务的突然死掉，通过分布式任务资源管理平台，在另外一台服务器上尝试重启任务。

**10. 概述总结**

    以上基础服务演变的概述比较粗糙，但是大致能够表明未来演变的核心方向和功能，这也是根据自身平台业务不同，方向不同形成的不同于常规开源解决方案的演变方向。当然纠结于细节实现的时候，也会比文中所述更加繁琐，功能更多，性能和实现要求更高，更偏向于轻量级和业务适应性。

五. 基础服务自主研发战略

   在网站研发处于前期或者创业未盈利阶段，可以考虑大量使用第三方的开源框架，以布局整体架构，及考虑架构的完整性和扩展性。虽然如此，但凡大型网站或者网站涉及到高并发，大流量的压力，核心的基础服务的基础设施必须全部或者部分自研。因为这种性能要求极高的网站，必须对各个细节的把控和要求都很严格，对基础服务的性能，质量要求也极高，采用一些完善的第三方开源框架反而可能是种累赘（如redis,leveldb等轻量级的除外）。而且未来基础服务的统一监控，弹性伸缩，及与云计算的层面的自主伸缩性契合都需要修改部分核心代码实现。如果采用第三方框架，必须对这些代码非常了解后修改，同时还要不断的跟开源社区版本保持分支一致。一般第三方开源框架往往是集大成的常规解决方案，更加通用化，而我们业务往往只需要一部分功能的轻量级解决方案足矣，性能更高。

    故而从短期看基础服务使用第三方可以快速的部署架构，从长期看基础服务未来必定需要改进或者自主研发。而且研发基础服务的技术难度，在后期做弹性基础服务和云计算平台时来说其实不算什么，反而是更好的技术沉淀和基石。*（目前淘宝，京东，美团，蘑菇街，大众点评，当当网，窝窝团，58同城等都采取部分或者全部自研基础服务的方式。）*

六. 基础服务开源战略

    公司的竞争一般在于商业本质的竞争，而非在于技术的竞争。故开源基础服务对于公司来说，若能形成开源生态圈，则可以促进开源项目稳定性，优化开源代码，根据反馈不断的提升自身的基础服务产品，吸引相关的高级技术人才维护检验项目，减少项目的开发维护成本，同时提升公司在技术领域的影响力，提升开发人员的成就感。*（目前淘宝，当当网，58同城等都有部分项目开源）*

***1）开源成长路线***：

路线1:下载开源源码->学习开源项目->成功部署项目（根据开源文档或者QQ群项目管理员协助）->成为QQ群相关项目管理员->了解并解决日常开源项目问题->总结并整理开源项目文档并分享给大家或推广->成为git项目的开发者和参与者

路线2:下载开源源码->学习开源项目->成功部署项目（根据开源文档或者QQ群项目管理员协助）->在实际使用中发现bug并提交bug给项目相关管理员

路线3:下载开源源码->学习开源项目->成功部署项目（根据开源文档或者QQ群项目管理员协助）->自行建立开源项目分支->提交分支新功能给项目官方开发人员->官方开发人员根据项目情况合并新功能并发布新版本

***2）关于开源生态圈的构想***

生态闭环：官方开源项目->第三方参与学习->第三方改进并提交新功能或bug->官方合并新功能或bug->官方发布新版本

为什么开源? .net 开源生态本身弱,而强大是你与我不断学习，点滴分享,相互协助，共同营造良好的.net生态环境。

开源理念: 开源是一种态度，分享是一种精神，学习仍需坚持，进步仍需努力，.net生态圈因你我更加美好

***by 车江毅***

*(仅根据实际业务所设想的基础服务演变方向，不包含分布式存储，搜索引擎，大数据等，欢迎交流)*

***开源QQ群: .net 开源基础服务  238543768***

# 【开源】.Net 分布式服务中心

[收藏](javascript:void(0);)

[车江毅](https://my.oschina.net/chejiangyi/home)

* 发表于 8个月前

* 阅读 1083

* 收藏 31

* 点赞 11

* [评论 5](https://my.oschina.net/chejiangyi/blog/632018#comment-list)

*摘要: 当垂直应用越来越多，应用之间交互不可避免，将核心业务抽取出来，作为独立的服务，逐渐形成稳定的服务中心，当前服务支持thrift服务框架。支持服务的注册发现，服务调用的负载均衡，服务调用的监控。*

## 分布式服务中心

**开源地址**： <http://git.oschina.net/chejiangyi/Dyd.BaseService.ServiceCenter> 

当垂直应用越来越多，应用之间交互不可避免，将核心业务抽取出来，作为独立的服务，逐渐形成稳定的服务中心，当前服务支持thrift服务框架。支持服务的注册发现，服务调用的负载均衡，服务调用的监控。

### 功能简述

#### 支持协议

前期支持Thrift，支持多种服务通信协议（thrift，自定义协议）

#### 服务注册发现

管理中心发布服务，服务节点自动注册节点信息。

#### 服务负载均衡

实现服务节点的。

#### 服务故障转移

自动移除故障服务节点。

#### 服务监控

提供服务节点Cpu占用，内存占用，线程，调用次数，连接数等相关数据的监控，记录出错日志。

#### 部署开发

项目部署，开发等相关文档见docs文件夹。

### 相关说明

未在正式项目中稳定使用（部分外围项目已使用），后期需要完善；仅供学习使用及优化，属于半研究性项目。  
希望该项目能够成为同类型优秀.Net项目的起点和启发;欢迎有兴趣的开发人员一起完善。  
**开源相关群: .net 开源基础服务 238543768**

### 开发人员

车江毅 (核心架构设计及编码)（因个人身体不适，后续未跟进测试调优性能及完善）  
盛海江 (部分核心编码，web管理功能，测试，文档)

### 部分界面截图

**开源地址**：<http://git.oschina.net/chejiangyi/Dyd.BaseService.ServiceCenter>

by 车江毅

# 【开源】.Net Api开放接口文档网站

[收藏](javascript:void(0);)

[车江毅](https://my.oschina.net/chejiangyi/home)

* 发表于 5个月前

* 阅读 379

* 收藏 4

* 点赞 0

* [评论 0](https://my.oschina.net/chejiangyi/blog/685933#comment-list)

*摘要: ApiView是.net api的接口文档查看网站,用于解决分布式开发过程中的Api接口管理和沟通问题。 自动生成api文档； 方便调试及第三方开发人员对接，可以应用在asp.net mvc，wcf，webservice 中使用； 代码及原理都很简单，方便修改和完善。*

开源地址:<http://git.oschina.net/chejiangyi/ApiView>

**开源QQ群: .net 开源基础服务  238543768**

**ApiView**

.net api的接口文档查看网站,用于解决[分布式开发](http://my.oschina.net/chejiangyi/blog/521950)过程中的Api接口管理和沟通问题。  
*- 自动生成api文档；  
- 方便api调试及第三方开发人员对接，可以应用在asp.net mvc，wcf，webservice 中使用；  
- 代码及原理都很简单，方便二次开发和完善。*

***by 车江毅***

**安装包**  
使用git下载项目并打开目录 “\安装包\” 可直接安装使用

**使用Demo示例**

1 using System;

2 using System.Collections.Generic;

3 using System.Linq;

4 using System.Web;

5 using System.Web.Mvc;

6 using BSF.BaseService.OpenApi.Attributes;

7 using BSF.Extensions;

8 using BSF.Api;

9

10 namespace ApiViewTestWeb.Controllers

11 {

12 public class DemoController:Controller

13 {

14 //

15 // GET: /Demo/

16

17 [OpenDoc("Index","test method","参数描述","结果描述","结果状态描述","作者",1.0,1.0,"备注")]

18 public string Index(int a,string b,float d)

19 {

20 return a+b.NullToEmpty()+d;

21 }

22

23 [OpenDoc("某一店铺的商品列表V15", "某一店铺的商品列表V15",

24 @"token:token(可选参数),

25 商户账号:shopid,

26 类目id:categoryid int?,

27 关键词:keyword,

28 条形码: barcode,

29 显示数量:pageSize ,

30 页码:pageIndex",

31 @"活动序号(Int):hdxh,

32 商品条码(string):sptm,

33 商品名称(string):spmc,

34 规格型号(string):ggxh,

35 计量单位(string):jldw,

36 商品图片(string):sptp,

37 图片修改时间(string):sptpxgsj,

38 原零售价(Decimal):original\_lsj,

39 零售价(Decimal):lsj,

40 商品销量(Decimal):spxl,

41 是否活动商品(int):isActivity,

42 活动总数量(Decimal):hdzsl,

43 每人限购数量(Decimal):mrxgsl,

44 ", "-5缺少参数", "车江毅", 1.5, 1.5, "")]

45 public ActionResult List(string shopid, int? categoryid, string keyword, string barcode, long? pageSize, long? pageIndex)

46 {

47 return Json( new ServiceResult() { code=1, data="测试数据", msg="成功", total=1 });

48 }

49 }

50 }

**OpenApi使用说明文档**

1.引用BSF.dll和BSF.BaseService.OpenApi.dll  
2.在MVC网站Controller下面的接口方法处添加OpenDocAttribute特性注释即可。  
3.参考ApiViewTestWeb使用方式中的DemoController。  
4.在AppView站点中发布OpenApi的信息。  
  配置AppView中web.config;配置如下：  
  <!--多个Api文档可以配置多个DllPath,如DllPath1,DllPath2;  
      接口dll路径 api程序集路径,多个;分隔 格式：需要公开接口的dll路径;接口名;测试的appurl'-->  
    <add key="DllPath1" value="D:\svn-working\RT\_Cloud\trunk\源代码\BaseService\ApiView\ApiViewTestWeb\bin\ApiViewTestWeb.dll;ApiViewTestWeb【示例】;http://10.17.72.96:8081/"/>  
5.点击AppView站点“帮助文档”中“重新加载”按钮生效。

**部分截图**

# 【开源】.Net 动态脚本引擎NScript

[收藏](javascript:void(0);)

[车江毅](https://my.oschina.net/chejiangyi/home)

* 发表于 4个月前

* 阅读 512

* 收藏 7

* 点赞 2

* [评论 3](https://my.oschina.net/chejiangyi/blog/700397#comment-list)

*摘要: .net 动态脚本引擎, 用于解决.net环境windows系统下类似java中Grovvy的功能和方向。 在互联网项目可以用来做一些功能，如动态营销活动（营销业务解耦和剥离），规则引擎，流程引擎，windows运维脚本，源码式插件开发。*

***开源地址***: <https://git.oschina.net/chejiangyi/NScript>

**开源QQ群: .net 开源基础服务  238543768**

**.Net 动态脚本引擎 NScript**

用于解决.net环境windows系统下类似java中Grovvy的功能和方向。  
在互联网项目可以用来做一些功能，如动态营销活动（营销业务解耦和剥离），规则引擎，流程引擎，windows运维脚本，源码式插件开发等。  
  
  
**使用方式**

 包括exe Main方式,程序集方式,应用程序域三种方式。

**最终编译文件**

  BSF.BaseService.NScript.exe

*exe 说明*  
*1) 可以用cmd命令运行本exe  
   /run 命令格式:/run {filename} {args}*

*说明:必须实现Main入口函数. {filename} 为文件路径,{args}为Main入口参数，默认空格分隔。  
   /help 命令格式:/help*

*说明:用户查看当前exe支持的命令说明。  
2）exe本身就是脚本编辑器。  
   可以用作.net 脚本的编辑工具，在实际运行环境中直接编辑或临时修改代码,开发环境中建议还是使用vs。  
3）exe本身也是脚本运行时。  
   整个脚本解析和运行时。  
4) exe大小  
   exe 本身很小很小，因为打包合并了第三方编辑器控件,才变更大些。*

*5）exe可以被解决方案以dll方式引用，并使用NScriptHelper接口方法。*

**Main方式示例**

*说明:*

*以exe命令的方式运行脚本示例  
1) 点击“运行”.bat  
2) 点击 "bsf.baseservice.nscript.exe" 进行脚本编辑及调试。*

*截图:*

*压缩包下载*:<http://share.weiyun.com/043fe46f11aabc1c2c17eb4d7cfa1e00>

**程序集或应用程序域方式运行示例**

*说明*  
*1) 解决方案引用"BSF.BaseService.NScript.exe"  
2) 程序集方式及应用程序域方式使用代码demo。*

{

this.richTextBox1.Text = @" public class B

{

//static void Main(string[] args)

//{

// System.Console.WriteLine(""hello"");

// System.Console.ReadLine();

// System.Console.ReadKey();

//}

public string test(string a)

{

return a;

}

}";

}

//程序集方式

private void button1\_Click(object sender, EventArgs e)

{

try

{

string code = this.richTextBox1.Text;

CompilerResult result = null;

var r = NScriptHelper.Run(new CompilerParams()

{

EnumSourceType = Core.EnumSourceType.Code,

EnumCompilerMode = Core.EnumCompilerMode.Assembly,

CodeOrFileName = code

},

"B", "test", new object[] { "a" }, out result);

MessageBox.Show(r);

}

catch (Exception exp)

{

MessageBox.Show(exp.Message);

}

}

//应用程序域方式

private void button2\_Click(object sender, EventArgs e)

{

try

{

string code = this.richTextBox1.Text;

CompilerResult result = null;

var r = NScriptHelper.Run(new CompilerParams()

{

EnumSourceType = Core.EnumSourceType.Code,

EnumCompilerMode = Core.EnumCompilerMode.AppDomian,

CodeOrFileName = code

},

"B", "test", new object[] { "a" }, out result);

MessageBox.Show(r);

}

catch (Exception exp)

{

MessageBox.Show(exp.Message);

}

}

*压缩包下载*：<http://share.weiyun.com/5b1f0adf5526b01c7f8ace09eaf9d113>

**使用exe编辑脚本示例**

**代码main.cs文件编写示例**

/\*

 \* codefiles=a.cs,codes\b.cs;//其他编译代码文件 ,分割多个文件 (支持相对路径) , 大小写敏感 （不要有分号和等号）

 \* dllfiles=System.dll;//引用的dll,即编译需要的dll ,分割多个dll (支持相对路径) ,大小写敏感 （不要有分号和等号）

 \* compilerlanguage=csharp;//编译语言类型,默认C#,可以不写

 \*/

/\*

\* 以上为主文件的编译头信息,必须要写置顶在代码文件头部。 包含源码文件信息,dll相关引用信息,代码编写语言;

      头信息中不要非常规的去使用;和=号，这个是用来解析的分隔符。主文件建议使用.main.cs命名结尾,这样可以自动识别,其他代码文件为cs结尾。

\* 以下为代码编码内容,语法遵循.net本身的语法及书写规范

\* by 车江毅

\*/

**using**System;

**using**System.Collections.Generic;

**using**System.Linq;

**using**System.Text;

**namespace**mytest //可以不要命名空间也可

{

**public**class B

    {

        //程序集或者应用程序域方式运行

        //关于调试: 通过“编辑器”->“调试” 暂不支持传入参数调试,但是实际环境是可以传入参数的

**public**string **test**()

        {

            return **new C**().**test**();

        }

        //Main编译方式需要指定的Main入口函数

        //关于调试: 通过“编辑器”->“调试” 暂不支持传入参数调试,但是实际环境是可以传入参数的

        static void **Main**(string[] args)

        {

            System.Console.**Read**();

        }

    }

}

**by 车江毅**

# .Net 分布式云平台基础服务建设说明概要

[收藏](javascript:void(0);)

[车江毅](https://my.oschina.net/chejiangyi/home)

* 发表于 4个月前

* 阅读 402

* 收藏 6

* 点赞 3

* [评论 1](https://my.oschina.net/chejiangyi/blog/710593#comment-list)

*摘要: 根据公司目前的业务情况，进行分布式云平台基础服务建设的架构，现状，取舍，概述以及展望。 包含数据库中间件，TCP服务框架，认证中心，服务中心，统一监控，配置中心，消息队列，任务调度平台，分布式缓存，文件服务，日志平台，开发接口平台，分布式部署平台，开发Api网关相关内容。*

**1）  背景**

建设云平台的基础框架，用于支持各类云服务的业务的构建及发展。

**2）  基础服务**

根据目前对业务的理解和发展方向，总结抽象出以下几个基础服务，如图所示

**3）  概要说明**

基础服务的发展会根据业务的发展，调整和完善，也会不断的改进，演变及完善；当然根据目前公司的现状和对基础服务的迫切程度，基础服务各模块的定位和发展预期将如下所述。

**1）     数据库中间件**

公司现状：

1）     对多种类型数据库的支持需求迫切，如同时支持mysql，orcale，sqlserver这些数据库。最多改动少量代码就可以在多种数据库类型中切换。

2）     尽量不要让开发人员写sql代码，因为原有的开发人员开发方式是采用linq的方式，大部分开发的业务是winform类型的业务。

采用方案：

目前采用entity framework，因entity framework 本身采用linq方式编程，自身能够解析linq为sql，且兼容多种数据库类型的查询。Entity framework 在.net 中的流行程度较高，代码开源，版本发展较快，且拥有大量应用文档和学习资料，相比较同类型的框架而言，当首选之。

方案弊端：

Entity framework 的采用只是临时的方案，因为业务的需要会持续比较久的时间。

1）     从高性能的服务来看，linq to sql的过程必然会有性能损失，即便框架做了解析的缓存，但是也无法避免这些问题。一些复杂语句的查询，linq to sql 目前也会出现意外的解析结果，复杂的语句查询难以用linq表达。如果不是对linq to sql 这种方式较熟练和关注性能的人，一般写法上也会导致性能问题。

2）     从数据库的角度看，目前业务暂时还使用同一个数据库，未来业务会采用多个数据库，多张数据表。这一点entity framework 后面会涉及到分库的支持和分表的支持，显然即便修改源码也很头疼。而且多个数据源，多个数据库类型的支持，意味着同一个业务要涉及到多种数据库下面性能的调优和运维，特别是涉及到版本升级的数据迁移，要兼容多种数据库，意味着工作量真心不小。

未来方向：

采用单一类型的数据库，会有一个支持sql编写直连数据库，支持分库分表的分布式数据库，自动管理数据库连接池，自动提供性能分析及预警等的数据库中间件。

**2）     TCP服务框架**

公司现状：

1）     用于采集程序，采集设备和服务器的直连，发送采集信息。

2）     服务器端反向通知连接程序或设备，即时通知信息。

3）     与工作站的通信环境（云平台采用ActiveMQ）,连接第三方设备（采用signalr asp.net）。

采用方案：

暂时保持与工作站的通信环境（云平台采用ActiveMQ）,连接第三方设备（采用signalr集成入asp.net）这种方案。因为公司目前采用C#编程，这两块技术选型都有相应的C#客户端或者C#的解决方案的一些示例；故使用起来问题应该不大，但是遇到的问题也不会少。若遇到性能问题，可以简单的通过扩展多个ActiveMQ和 应用服务的负载均衡来解决。

其他方案：

采用redis或者rabbitmq之类的类似解决方案，个人倾向与redis 的发布订阅机制解决，性能不算差（听闻过上规模的使用案例及跨语言客户端sdk）（且可以统一缓存的使用框架，便于维护。）

方案弊端：

1）     无论采用redis，activemq，rabbitmq之 类的哪种消息队列方式解决，都无法避免本质的性能问题，因为这些框架本身是用来解决消息队列的，因为其内存消息转发机制，故而用于一些即时通讯，常用于解 决内网环境的一些应用交互。目前的场景会应用到广域网环境与工作站进行通信，业内类似的解决方案也曾有人使用过，但是也会经常出现activemq 内存满或者莫名死掉的情况。

2）     采用signalr 应用挂载到asp.net 上面的使用方式，经过一些第三者开发的经验，也会出现稍微高并发就出现性能问题或者死掉的情况。Signalr 常用于.net 技术，也有简单的使用案列，但是还没有成熟的上规模的使用案例和场景。

未来方向：

采用java的NIO思想或者Windows 完成端口思想,搭建纯粹的TCP socket服务是解决本质的一个方案，一般一台服务器能够承载10万的连接，几千的活动连接（具体看服务器配置等情况）不会有问题（而旧方案可能承载几千，上百的活动连接就会出现性能问题）。

**3）     认证中心**

公司现状：

1）     原有工作站内网子系统的登陆验证，外网设备登录验证，云平台用户登录验证。

2）     云平台用户菜单权限获取，操作权限获取。

采用方案：

自行研发公司特有业务的认证中心平台，目前仅第一个版本。包含

1）     设备管理，子系统管理，云平台用户管理和权限管理等

2）     第三方使用的登录接口，用户菜单权限接口，用户操作权限接口。

以上功能目前能够满足现有公司的业务。

方案弊端：

1）     目前比较简单，通过token授权，无名加密，无appid和serect秘钥授权之类的。故而没有较强的安全机制，但是能够满足实际开发。而且目前的公司业务对于安全的要求并不高。

2）     通信性能不高，因为目前采用Http协议进行通信，本身通信性能不高。而且认证中心将承载所有业务的认证，基本上所有云项目模块等业务都会将请求汇聚到认证中心的接口上，在后续公司流量的发展上必然会出现第一个出现接口上的性能问题。

未来方向：

1）     平台所有的接口实现内部必须有redis缓存，平台接口客户端使用要有sdk封装（在sdk内部做客户端缓存，哪怕默认5 s的缓存）

2）     平台的所有接口后续接到“高性能服务中心”，走TCP连接池的通信方式实现，提高内部通信的性能。

**4）     服务中心（个人开源地址：**[**http://git.oschina.net/chejiangyi/Dyd.BaseService.ServiceCenter**](http://git.oschina.net/chejiangyi/Dyd.BaseService.ServiceCenter)**）**

公司现状：

1）     项目之间出现互相调用的业务耦合，目前采用dll的方式调用，但是出现dll更新出错及管理等情况，导致开发人员认为效率不高。

2）     公司迫切希望采用微服务/soa的架构方式来剥离项目的业务耦合，简化上下游业务调用的管理方式。

采用方案：

1）     暂时采用Http restful类似的方式提供服务化的接口，供第三方接口调用，同时这也符合soa服务化的架构思想。

2）     通过api view 自动公开接口文档，上下游之间调用调试，方便开发人员沟通协调。

方案弊端：

1）     个人经验：服务化的接口方式有效的,对业务沟通也是非常有帮助的，但是未必能够真正的在效率上得到本质的提升。但是对于项目的模块化管理应该有较好的帮助。

2）     Http的接口通信方式效率并不高，作为服务框架必然是走TCP的内部通信，性能才会有明显提升。

3）     服务治理，协调方面的问题为考虑，如没有考虑调用链死循环，调用链上的性能导致雪崩，上下游服务监控，上下游客户端服务变更历史记录及通知，上下游客户端服务协议耦合剥离，服务化层面的负载均衡和故障转移等等众多问题。

未来方向：

1）     自研服务中心，将性能，服务治理，协调等工作从业务开发中抽离抽象出来，业务开发只需要关注无状态的业务服务开发即可。

2）     所有内部的业务全部剥离（不仅仅是耦合的业务），迁移到内部的服务中心，如果内部服务需要对第三方公开，可以提供Http的开放网关服务进行调用，网关层会做一些授权管理等工作，网关自身做负载均衡。

**5）     统一监控（个人开源地址：**[**http://git.oschina.net/chejiangyi/Dyd.BaseService.Monitor**](http://git.oschina.net/chejiangyi/Dyd.BaseService.Monitor)**）**

公司现状：

1）     项目处于前期研发阶段，没有较大规模的服务器集群，没有遇到多版本接口兼容，没有遇到线上监控问题和线上排查问题，性能问题的痛苦，对这些情况还不了解和敏感。

2）     开发人员希望解决项目开发调试时候，错误日志及错误日志的堆栈问题，调用路径问题排查。

采用方案：

1）     采用Http Restful 服务化业务接口的方式，应该能缓解项目开发调试的问题。（开发调试问题以前没遇到过，应该跟原来架构和技术采用wcf等方式有关导致）

2）     搭建分布式监控平台，因为是本人已有开源的项目，使用起来问题不大。能够解决很多云上服务器管理，性能监控及预警，sql性能监控，api接口性能监控，统一错误日志等。

方案弊端：

1）     个人还不是特别确切了解目前项目开发人员调试项目开发过程中，对日志问题真正迫切的本质原因，也没深刻体验（一直开发以来没有遇到问题难调试的问题，可能现有公司项目架构方式关系密切），所以不知道是否能够解决。

2）     目前分布式监控平台是在原有公司开发的简化版本，为了实现整体项目架构的监控那块的抽象和布局而研发的。性能和功能上还有很多的优化和改进空间。（当然支持公司的现状还是绰绰有余）

未来方向：

1）     根据公司的业务对监控的需求，还需要不断的改进和完善监控平台。

2）     监控平台的功能和性能需要完善，底层将使用nosql来存储实现。

**6）     配置中心（个人开源地址：**[**http://git.oschina.net/chejiangyi/Dyd.BaseService.ConfigManager**](http://git.oschina.net/chejiangyi/Dyd.BaseService.ConfigManager)**）**

公司现状：

1）     目前公司有类似配置中心的功能，用于基本的业务配置的使用，比较简单。

2）     云这块业务尚处于简单的业务模型和业务状态，未遇到真正线上复杂的业务和业务剥离的需求，及异步化的功能点，统计类的功能等等，对分布式配置中心的本质需求和问题还没有真正暴露出来。

采用方案：

依旧使用原有的配置中心功能，同时分布式的配置中心也会搭建。原有的配置中心适合业务配置的存储，现有的配置中心可以用于业务配置的存储，也可以用于分布式架构的环境配置协调问题。

方案弊端：

会维持两套配置中心的运维，在业务架构上比较难以区别，业务上容易混乱。

未来方向：

1）     两块配置中心将根据业务的需求和方向，在一定程度上进行融合。但就目前的公司精力不会着重这块。

2）     配置中心将根据公司的业务发展，也会继续演变出更多的功能，不过暂时不明朗。

**7）     消息队列（个人开源地址：**[**http://git.oschina.net/chejiangyi/Dyd.BusinessMQ**](http://git.oschina.net/chejiangyi/Dyd.BusinessMQ)**）**

公司现状：

1）     目前公司在云平台端与工作站异步通信是通过ActiveMQ进行的。

2）     云平台项目还处于前期研发起步阶段，业务复杂度还不够，对性能的要求不高，也未涉及同一业务异步化拆分和解耦。

3）     公司的采集方面的业务还未做到真正的大规模分析，大规模采集的场景。

采用方案：

出于公司架构统一的现状考虑，暂时采用ActiveMQ，也方便java，C#等跨语言的异步通信。当然也仅仅能应用与异步化的简单的即时通信效果。

方案弊端：

ActiveMQ 只能作为异步的即时通信暂时使用，就目前的性能和稳定性来说，并不是长远之计。

1）     若是为了持久化的Tcp通信，未来自己会有TCP服务的搭建来确保。

2）     若是为了消息队列的通信，未来更多考虑消息的堆积性能，消息的高稳定性和高可靠性（不能丢失消息）。

3）     若是考虑消息队列收集消息便于后续采集分析，未来更多考虑类似kafka的方案。

未来发展：

消息队列有众多的解决方案，也有众多的一些特性适用于不同的业务场景。针对这些不同的场景和方案，个人会做如下考虑：

1）     自建的一套消息队列平台，自建的消息队列可以剥离底层的存储引擎，通过不同的存储引擎的特性，达到适用不同场景和方案的目的。（如存储引擎为redis，ssdb，数据库等，即便实现逻辑相同，但是性能不同，可靠性表现也不同）

2）     自建的一套消息队列中间件，可以剥离具体的消息队列实现，抽象出常规消息队列的使用方式。仅通过修改连接字符串或者配置类，就能实现不同消息平台的切换。（如底层消息服务可能是activemq，rabbitmq，redis，kafka，对上层业务可以是透明，甚至无缝切换）

**8）     任务调度平台（个人开源地址：**[**http://git.oschina.net/chejiangyi/Dyd.BaseService.TaskManager**](http://git.oschina.net/chejiangyi/Dyd.BaseService.TaskManager)**）**

公司现状：

1）     公司目前业务尚处于前期，未有业务需求有类似后台任务统计，后台任务消费之类的业务需求。

2）     任务调度平台是所有基础服务的一个基础环节，目前也仅在基础服务部署中使用。

采用方案：

个人开源的分布式任务调度平台。

方案弊端：

分布式任务调度平台目前仅属于一个简单的任务调度平台，未来的发展方向还有很大的空间。

未来发展：

1）     所有公司的业务都被视为一个业务任务，所有的业务任务都将被挂载到任务调度平台，任务调度平台会根据分布式集群的负载情况，自动分配集群服务器用于业务的负载均衡和故障转移等资源的调度和协调。（如所有的接口服务，所有的后台任务，所有的消息消费任务等等）

2）     任务调度平台也可称为类似于hadoop之类的大数据处理，实时计算平台，用于批量处理实时的，非实时的一些动态的流式的任务创建，回收，协调。（如类似爬虫之类的采集业务，和算法分析任务等）

**9）     分布式缓存（个人开源地址：**[**http://git.oschina.net/chejiangyi/XXF.BaseService.DistributedCache**](http://git.oschina.net/chejiangyi/XXF.BaseService.DistributedCache)**）**

公司现状：

1）     目前公司的业务还不需要用到分布式缓存的使用，除了认证中心这块应该在后续涉及到一些性能问题。

采用方案：

个人开源的分布式缓存中间件。（目前实现的是基于memcached协议的一个统一的分布式缓存框架）

方案弊端：

仅支持基础的分布式缓存框架，整体数据结构比较简单（key+定时过期失效），但是也是缓存中最实用的功能。

未来发展：

1）     支持更多的协议，如redis的通信协议。及更多底层存储框架的抽象。（每种缓存框架都有特定的使用场景和微妙的差别）

2）     分布式缓存的统一性能监控；一致性哈希的支持便于实现定制的故障转移方案，避免雪崩等缓存失效场景。

3）     根据公司的业务支持其他缓存场景，如本地缓存一致性（协同分布式消息队列实现）的支持。

**10）  文件服务**

公司现状：

1）     公司目前的采集的业务将信息都存在本地的应用服务器，以文件形式存储。

2）     公司的采集业务信息文件，需要持久保存。

采用方案：

暂时保持现状。

方案弊端：

1）     无论从容量的扩容和性能的角度看，单独的文件服务器是一个长远的必然需求。

2）     目前的业务可能涉及到文件的连续存储和文件部分内容的读取的需求，就市面上的开源文件服务可能不满足需求。

3）     个人现在对公司关于文件服务的业务需求，还不是特别了解。

未来发展：

1）     考虑自研分布式文件服务，读取性能未必胜于市面的开源文件服务。（自研的文件服务应该还是基于windows文件管理结构），但是灵活度会更高。

2）     自研的分布式文件服务sdk，要在使用上抽象，并兼容公司的底层存储差异（有些大文件存储可能还是使用第三方存储，但是对于开发来说是透明无感知的）。

**11）  日志平台**

公司现状：

1）     公司目前对于项目调试的困难，导致对日志平台的需求。

2）     公司的业务暂时还不需要基于日志的分析需求，对于大容量日志的记录及基于日志的堆栈调用链记录需求。

采用方案：

暂时通过监控平台的错误日志和本地的错误日志打印，解决目前对错误调试的需求。

监控平台也支持常规业务日志的打印，但是此业务日志的打印不支持大容量的需求。（过多打印会造成自身程序阻塞）

方案弊端：

1）     监控平台也支持常规业务日志的打印，但是此业务日志的打印不支持大容量的需求。（过多打印会造成自身程序阻塞）。

2）     不支持调用链的日志记录及分析。不支持大容量的日志记录，不支持日志的毫秒级查找，便于问题定位。

未来发展：

1）     日志平台未来会自行研发（或者结合第三方开源），类似于目前开源的elk。平台的定位是大容量日志的收集，挖掘，分析，排查。

2）     更多的是结合自身的业务，和对未来业务发展的规划，对于日志平台的基础功能做特定的功能或者统计报表展现。

**12）  开放接口平台（个人开源地址：**[**http://git.oschina.net/chejiangyi/ApiView**](http://git.oschina.net/chejiangyi/ApiView)**）**

公司现状：

1）     公司的业务急切需要通过soa/微服务的方式提供接口，供第三方开发人员使用。

2）     Soa业务上下游之间需要维护文档，便于沟通和调试。

采用方案：

个人开源的appview 开放接口平台。类似swagger。

方案弊端：

目前开放接口平台实现很简单，功能也非常精简通用。还欠缺一些管理功能，比如版本变更记录和上下游版本变更通知等。

未来发展：

1）     开放接口平台会根据公司实际的问题和需求不断的完善功能，如根据公司的接口约定，自动检测并提醒不规范的接口。自动记录版本变更，自动邮件通知下游调用方接口变更，自动化的接口治理等功能。

**13）  分布式部署平台**

公司现状：

1）     公司的云平台业务尚在初期，流量远远没有上来，也没有任何性能问题。

2）     云平台的部署还没有考虑到分布式部署发布和运维的问题，也没有秒级全平台部署，版本管理，版本回滚的需求。

采用方案：

暂时前提先考虑人工多服务器发布解决。

方案弊端：

人工解决，在真实环境中往往出很多问题，毕竟人是最容易犯错的。所以公司上轨道后，往往采用全自动部署发布的问题。

未来发展：

1）     自研一套分布式部署发布的平台，做到版本管理，异常回滚，分布式部署等问题。（这个实现并不复杂，够用即可）

**14）  开放Api网关**

公司现状：

1)        公司目前采用WCF的方式公开服务，调试和使用都很麻烦。

采用方案：

1）     即将采用http 直接公开soa业务服务的方式解决问题，这种方式粗暴但也非常有效。

2）     后面服务中心开发稳定后，所有业务将会迁移到服务中心，所有业务通过tcp连接池访问，提高通信效率，从而提升性能和响应时间。

方案弊端：

1）     第三方开发人员想通过第三方api访问，则往往不支持。

未来发展：

1）     开放api网关，将所有内网的服务api，对可以通过Http的形式进行转发访问，Http网关和服务中心保持高性能通信。

2）     开放api网关遇到性能问题，则负载均衡即可。

3）     开放api网关将管理对外开放的api授权问题，api访问频率控制，api访问权限控制，api访问的协议控制（xml或者json等）。

剥离开放api管理的功能和api的具体业务实现。

**4）  总结**

由于时间的预算有限，以上内容均是对于目前基础服务各个平台的定位和架构方向的粗略阐述，也没有对文字重新校对；

因为未来业务的发展往往是多变的，故而基础服务的功能和方向也会不断的微调，但是总体的方向应该不会有所改变。希望粗略的文档能够让大家理解公司业务架构上的取舍和未来的演变方向。

By 车江毅

（备注：欢迎大家一起交流，分享，并指出架构的不足，tks！）

**开源QQ群: .net 开源基础服务  238543768**

# 【开源】.Net Aop(静态织入)框架 BSF.Aop

[收藏](javascript:void(0);)

[车江毅](https://my.oschina.net/chejiangyi/home)

* 发表于 2个月前

* 阅读 323

* 收藏 3

* 点赞 0

* [评论 0](https://my.oschina.net/chejiangyi/blog/739437#comment-list)

*摘要: .Net 免费开源，静态Aop织入（直接修改IL中间语言）框架，类似PostSharp(收费)； 实现前后Aop切面和INotifyPropertyChanged注入方式。*

## ****BSF.Aop****

.Net **免费开源，静态Aop织入**（直接修改IL中间语言）框架，类似PostSharp(收费)；  
实现前后Aop切面和INotifyPropertyChanged注入方式。

开源地址:<https://git.oschina.net/chejiangyi/BSF.Aop>

**开源QQ群: .net 开源基础服务  238543768  欢迎交流**  
  
描述：

* 静态织入实现，性能几乎接近原生代码编写。
* 项目结构扩展自BSF基础架构，但dll方面并不依赖BSF.dll，可以独立开发使用。
* 项目代码量少（也就几个类文件），核心功能插件化开发，便于第三方扩展，阅读，调试，一起来完善。
* .net Aop静态织入相关的免费开源项目比较少或暂未听闻，故希望开源这块内容，成为同类型开源项目的起点和借鉴。

#### ****使用配置****

1）项目引用BSF.Aop.dll,引用Mono.Cecil.dll，引用Mono.Cecil.Pdb.dll。  
2）项目启动代码添加AopStartLoader.Start();一句代码即可。  
（该代码用于自动注入扫描和vs项目环境自动配置,导出相关exe文件等）

备注：

* Web项目在Application\_Start中添加
* Winform项目在Pragram.Main中添加

##### ****自动化配置（默认推荐）****

Build项目，然后直接运行调试项目。

备注：

* AopStartLoader.Start();这句代码第一次项目运行会报错，因为Aop环境未曾自动搭建；第一次运行并自动搭建成功后，以后运行都是自动注入的。
* 默认会在项目下生成“packages.BSF.Aop”文件夹,并在vs项目PostBuildEvent事件中注入“aop运行脚本”，具体参考以下“手工配置内容”。

##### ****手工配置 （遇到问题，推荐使用）****

1）配置Aop注入目录。  
vs项目下新建packages.BSF.Aop 目录，里面分别包含BSF.Aop.ILRun.exe，BSF.Aop.dll，Mono.Cecil.Pdb.dll,Mono.Cecil.dll 这几个文件。  
2）配置PostBuildEvent 脚本。

##### winform环境****:****

打开vs-》项目属性-》PostBuildEvent，配置aop运行脚本。如：  
xcopy $(OutDir)BSF.Aop.dll $(ProjectDir)packages.BSF.Aop\ /Y  
call "$(ProjectDir)packages.BSF.Aop\BSF.Aop.ILRun.exe" msgbox $(TargetDir)

##### web环境****:****

打开vs-》项目属性-》PostBuildEvent，配置aop运行脚本。(OpenApi.Test.Web替换成具体的项目名) 如：  
xcopy $(SolutionDir)\OpenApi.Test.Web\bin\BSF.Aop.dll $(SolutionDir)\OpenApi.Test.Web\packages.BSF.Aop\ /Y  
call "$(SolutionDir)\OpenApi.Test.Web\packages.BSF.Aop\BSF.Aop.ILRun.exe" msgbox $(SolutionDir)\OpenApi.Test.Web\bin\

#### ****使用demo示例****

1) 前后Aop切面示例 （详细参考BSF.Aop.Test项目）

public class AroundAopTest

{

[MyAroundAop]

[AttributeInfo(Des = "测试2")]

public void Method(TempInfo info, out int b,int a=1)

{

a = 222;

b = 3;

System.Console.WriteLine("Hello world!"+a);

}

}

public static class AroundAopTest2

{

[MyAroundAop][AttributeInfo(Des ="测试")]

public static void Method2(TempInfo info, int a = 1)

{

a = 222;

System.Console.WriteLine("Hello world!" + a);

}

}

public class MyAroundAop : Aop.Attributes.Around.AroundAopAttribute

{

public MyAroundAop()

{

}

public override void Before(AroundInfo info)

{

var att = info.Method.CustomAttributes.ToList()[0];

info.Params["a"] = 55;

System.Console.WriteLine("before" + info.Params["a"]);

}

public override void After(AroundInfo info)

{

System.Console.WriteLine("after"+ info.Params["a"]);

}

}

public class TempInfo

{

public int T1 { get; set; }

}

public class AttributeInfo : System.Attribute

{

public string Des { get; set; }

}

2) INotifyPropertyChanged 示例（暂未测试真正使用效果,详细参考BSF.Aop.Test项目 ）

[NotifyPropertyChangedAop]

public class User

{

public string Name { get; set; }

public int Age { get; set; }

[NoAop]

public int B { get; set; }

}

by 车江毅