特别专题 语言 & 开发

从百度文件系统看大型分布式系统设计中的定式与创新

百度的核心业务和数据库系统都依赖分布式文件系统作为底层存储，文件系统的可用性和性能对上层搜索业务的稳定性与效果有着至关重要的影响。 现有的分布式文件系统（如 HDFS 等）是为离线批处理设计的，无法在保证高吞吐的情况下做到低延迟和持续可用，所以我们从搜索的业务特点出发，设计了百度文件系统。 在百度文件系统设计中，一方面，涉及从中心化和对等模型的折衷，到元数据的扩展性，再到网络拓扑的选择等，一系...

浏览所有 **语言 & 开发**

- 架构 & 设计
  - 架构

- 企业架构
- 性能和可伸缩性
- 设计
- 案例分析
- 设计模式
- 安全

特别专题 架构 & 设计

从百度文件系统看大型分布式系统设计中的定式与创新


百度的核心业务和数据库系统都依赖分布式文件系统作为底层存储，文件系统的可用性和性能对上层搜索业务的稳定性与效果有着至关重要的影响。 现有的分布式文件系统（如 HDFS 等）是为离线批处理设计的，无法在保证高吞吐的情况下做到低延迟和持续可用，所以我们从搜索的业务特点出发，设计了百度文件系统。 在百度文件系统设计中，一方面，涉及从中心化和对等模型的折衷，到元数据的扩展性，再到网络拓扑的选择等，一系...

浏览所有 **架构 & 设计**

- 数据科学
  - 大数据
  - NoSQL
  - 数据库

特别专题 数据科学

Apache Beam实战指南之基础入门


本文是 Apache Beam 实战指南系列文章 的第一篇内容，将简要介绍 Apache Beam 的发展历史、应用场景、模型和运行流程、SDKs ，并结合 Beam 的应用示例和代码剖析带你进一步了解 Beam 的运用原理。

浏览所有 **数据科学**

- 文化 & 方法
  - Agile
  - 领导能力
  - 团队协作
  - 测试
  - 用户体验

- Scrum
- 精益

特别专题 文化 & 方法

汉堡屋的故事：从系统思维、瓶颈到跨职能


一家坐落在里约热内卢狭窄街道中的高档小汉堡屋开张了，他们的系统针对高效处理订单进行了优化，然而不幸的是，这种优化带来了混乱。有一天早晨，收银员没有上班，你能猜到发生了什么事儿吗？在“约束理论和系统思维”的帮助下，我们将在本文中解释为什么他们的系统确实改善了一个人的“短板”？

浏览所有 **文化 & 方法**

- DevOps
  - 持续交付
  - 自动化操作
  - 云计算

特别专题 DevOps

Steve Thair谈DevOps on Windows的演变与面临的挑战

InfoQ采访了DevOpsGuys的联合创始人Steve Thair，了解DevOps on Windows的演变、现状和面临的挑战。

浏览所有 **DevOps**

架构
移动
运维
云计算
AI
大数据
容器
前端
QCon
ArchSummit

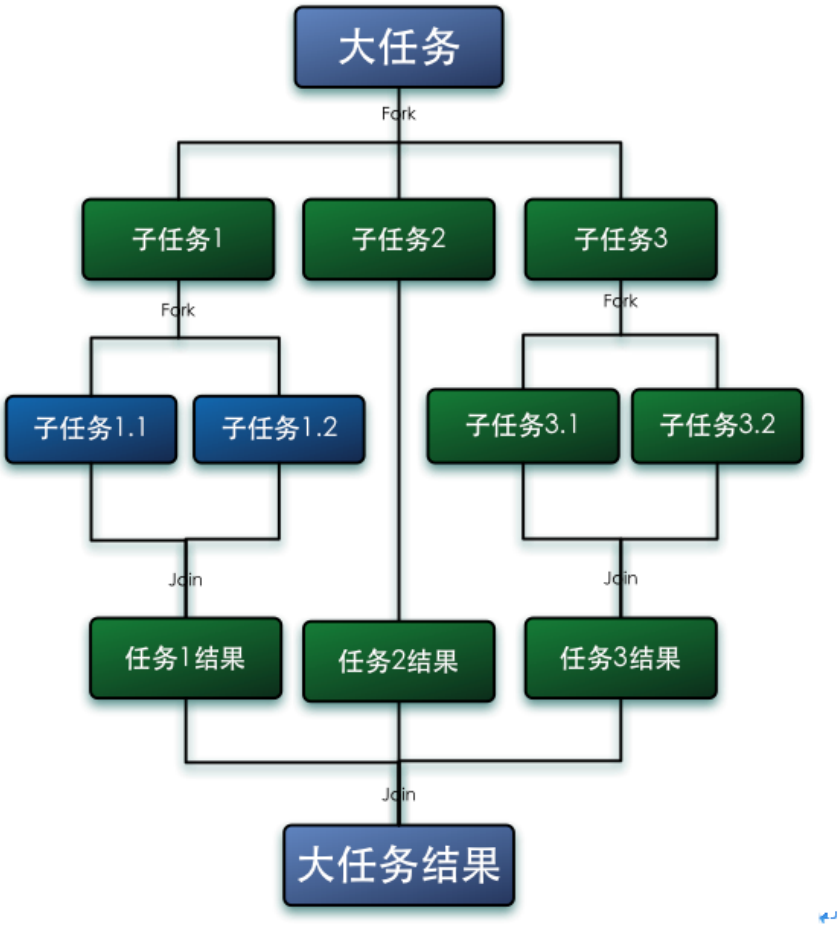
聊聊并发（八）——Fork/Join框架介绍

/作者 方腾飞 发布于 2013年12月23日. 估计阅读时间: 14 分钟 /硅谷人工智能、机器学习、互联网金融、未来移动技术架构 ，尽在QCon上海2017 2 讨论

1. 什么是Fork/Join框架

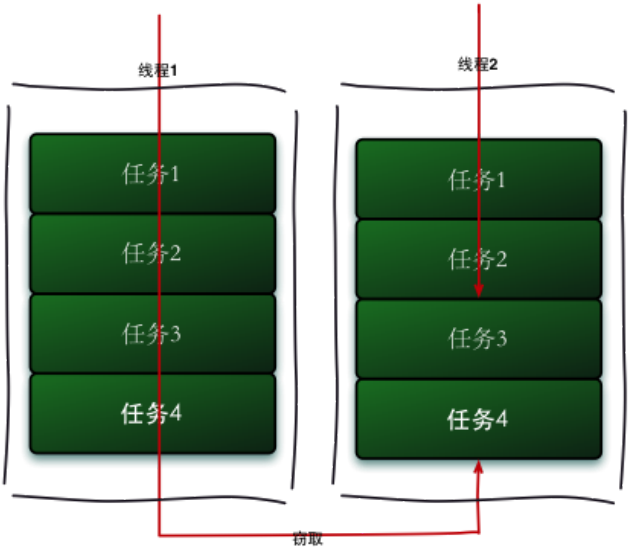
Fork/Join框架是Java7提供了的一个用于并行执行任务的框架， 是一个把大任务分割成若干个小任务，最终汇总每个小任务结果后得到大任务结果的框架。

我们再通过Fork和Join这两个单词来理解下Fork/Join框架，Fork就是把一个大任务切分为若干子任务并行的执行，Join就是合并这些子任务的执行结果，最后得到这个大任务的结果。比如计算1+2+。。+10000，可以分割成10个子任务，每个子任务分别对1000个数进行求和，最终汇总这10个子任务的结果。Fork/Join的运行流程图如下：



2. 工作窃取算法

工作窃取（work-stealing）算法是指某个线程从其他队列里窃取任务来执行。工作窃取的运行流程图如下：



那么为什么需要使用工作窃取算法呢？假如我们需要做一个比较大的任务，我们可以把这个任务分割为若干互不依赖的子任务，为了减少线程间的竞争，于是把这些子任务分别放到不同的队列里，并为每个队列创建一个单独的线程来执行队列里的任务，线程和队列一一对应，比如A线程负责处理A队列里的任务。但是有的线程会先把自己队列里的任务干完，而其他线程对应的队列里还有任务等待处理。干完活的线程与其等着，不如去帮其他线程干活，于是它就去其他线程的队列里窃取一个任务来执行。而在这时它们会访问同一个队列，所以为了减少窃取任务线程和被窃取任务线程之间的竞争，通常会使用双端队列，被窃取任务线程永远从双端队列的头部拿任务执行，而窃取任务的线程永远从双端队列的尾部拿任务执行。

工作窃取算法的优点是充分利用线程进行并行计算，并减少了线程间的竞争，其缺点是在某些情况下还是存在竞争，比如双端队列里只有一个任务时。并且消耗了更多的系统资源，比如创建多个线程和多个双端队列。

3. Fork/Join框架的介绍

我们已经很清楚Fork/Join框架的需求了，那么我们可以思考一下，如果让我们来设计一个Fork/Join框架，该如何设计？这个思考有助于你理解Fork/Join框架的设计。

第一步分割任务。首先我们需要有一个fork类来把大任务分割成子任务，有可能子任务还是很大，所以还需要不停的分割，直到分割出的子任务足够小。

第二步执行任务并合并结果。分割的子任务分别放在双端队列里，然后几个启动线程分别从双端队列里获取任务执行。子任务执行完的结果都统一放在一个队列里，启动一个线程从队列里拿数据，然后合并这些数据。

Fork/Join使用两个类来完成以上两件事情：

- ForkJoinTask：我们要使用ForkJoin框架，必须首先创建一个ForkJoin任务。它提供在任务中执行fork()和join()操作的机制，通常情况下我们不需要直接继承ForkJoinTask类，而只需要继承它的子类，Fork/Join框架提供了以下两个子类：
  - RecursiveAction：用于没有返回结果的任务。
  - RecursiveTask ：用于有返回结果的任务。
- ForkJoinPool ：ForkJoinTask需要通过ForkJoinPool来执行，任务分割出的子任务会添加到当前工作线程所维护的双端队列中，进入队列的头部。当一个工作线程的队列里暂时没有任务时，它会随机从其他工作线程的队列的尾部获取一个任务。

4. 使用Fork/Join框架

让我们通过一个简单的需求来使用下Fork / Join框架，需求是：计算1+2+3+4的结果。

使用Fork / Join框架首先要考虑到的是如何分割任务，如果我们希望每个子任务最多执行两个数的相加，那么我们设置分割的阈值是2，由于是4个数字相加，所以Fork / Join框架会把这个任务fork成两个子任务，子任务一负责计算1+2，子任务二负责计算3+4，然后再join两个子任务的结果。

因为是有结果的任务，所以必须继承RecursiveTask，实现代码如下：

```
package fj;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.Future;
import java.util.concurrent.RecursiveTask;

public class CountTask extends RecursiveTask<Integer> {

	private static final int THRESHOLD = 2;// 阈值
	private int start;
	private int end;

	public CountTask(int start, int end) {
		this.start = start;
		this.end = end;
	}
	@Override
	protected Integer compute() {
		int sum = 0;

		// 如果任务足够小就计算任务
		boolean canCompute = (end - start) <= THRESHOLD;
		if (canCompute) {
			for (int i = start; i <= end; i++) {
				sum += i;
			}
		} else {
			// 如果任务大于阈值，就分裂成两个子任务计算
			int middle = (start + end) / 2;
			CountTask leftTask = new CountTask(start, middle);
			CountTask rightTask = new CountTask(middle + 1, end);
			//执行子任务
			leftTask.fork();
			rightTask.fork();
			//等待子任务执行完，并得到其结果

			int leftResult=leftTask.join();
			int rightResult=rightTask.join();
			//合并子任务
			sum = leftResult  + rightResult;
		}
		return sum;
	}

	public static void main(String[] args) {
		ForkJoinPool forkJoinPool = new ForkJoinPool();
		// 生成一个计算任务，负责计算1+2+3+4
		CountTask task = new CountTask(1, 4);
		// 执行一个任务
		Future<Integer> result = forkJoinPool.submit(task);
		try {
			System.out.println(result.get());
		} catch (InterruptedException e) {
		} catch (ExecutionException e) {
		}
	}
}
```

通过这个例子让我们再来进一步了解ForkJoinTask，ForkJoinTask与一般的任务的主要区别在于它需要实现compute方法，在这个方法里，首先需要判断任务是否足够小，如果足够小就直接执行任务。如果不足够小，就必须分割成两个子任务，每个子任务在调用fork方法时，又会进入compute方法，看看当前子任务是否需要继续分割成孙任务，如果不需要继续分割，则执行当前子任务并返回结果。使用join方法会等待子任务执行完并得到其结果。

5. Fork/Join框架的异常处理

ForkJoinTask在执行的时候可能会抛出异常，但是我们没办法在主线程里直接捕获异常，所以ForkJoinTask提供了isCompletedAbnormally()方法来检查任务是否已经抛出异常或已经被取消了，并且可以通过ForkJoinTask的getException方法获取异常。使用如下代码：

```
if(task.isCompletedAbnormally())
{
    System.out.println(task.getException());
}
```

getException方法返回Throwable对象，如果任务被取消了则返回CancellationException。如果任务没有完成或者没有抛出异常则返回null。

6. Fork/Join框架的实现原理

ForkJoinPool由ForkJoinTask数组和ForkJoinWorkerThread数组组成，ForkJoinTask数组负责存放程序提交给ForkJoinPool的任务，而ForkJoinWorkerThread数组负责执行这些任务。

ForkJoinTask的fork方法实现原理。当我们调用ForkJoinTask的fork方法时，程序会调用ForkJoinWorkerThread的pushTask方法异步的执行这个任务，然后立即返回结果。代码如下：

```
public final ForkJoinTask fork() {         ((ForkJoinWorkerThread) Thread.currentThread())             .pushTask(this);         return this; }
```

pushTask方法把当前任务存放在ForkJoinTask 数组queue里。然后再调用ForkJoinPool的signalWork()方法唤醒或创建一个工作线程来执行任务。代码如下：

```
final void pushTask(ForkJoinTask t) {
        ForkJoinTask[] q; int s, m;
        if ((q = queue) != null) {    // ignore if queue removed
            long u = (((s = queueTop) & (m = q.length - 1)) << ASHIFT) + ABASE;
            UNSAFE.putOrderedObject(q, u, t);
            queueTop = s + 1;         // or use putOrderedInt
            if ((s -= queueBase) <= 2)
                pool.signalWork();
        else if (s == m)
                growQueue();
        }
    }
```

ForkJoinTask的join方法实现原理。Join方法的主要作用是阻塞当前线程并等待获取结果。让我们一起看看ForkJoinTask的join方法的实现，代码如下：

```
public final V join() {
        if (doJoin() != NORMAL)
            return reportResult();
        else
            return getRawResult();
}
private V reportResult() {
        int s; Throwable ex;
        if ((s = status) == CANCELLED)
            throw new CancellationException();
if (s == EXCEPTIONAL && (ex = getThrowableException()) != null)
            UNSAFE.throwException(ex);
        return getRawResult();
}
```

首先，它调用了doJoin()方法，通过doJoin()方法得到当前任务的状态来判断返回什么结果，任务状态有四种：已完成（NORMAL），被取消（CANCELLED），信号（SIGNAL）和出现异常（EXCEPTIONAL）。

- 如果任务状态是已完成，则直接返回任务结果。
- 如果任务状态是被取消，则直接抛出CancellationException。
- 如果任务状态是抛出异常，则直接抛出对应的异常。

让我们再来分析下doJoin()方法的实现代码：

```
private int doJoin() {
        Thread t; ForkJoinWorkerThread w; int s; boolean completed;
        if ((t = Thread.currentThread()) instanceof ForkJoinWorkerThread) {
            if ((s = status) < 0)
 return s;
            if ((w = (ForkJoinWorkerThread)t).unpushTask(this)) {
                try {
                    completed = exec();
                } catch (Throwable rex) {
                    return setExceptionalCompletion(rex);
                }
                if (completed)
                    return setCompletion(NORMAL);
            }
            return w.joinTask(this);
        }
        else
            return externalAwaitDone();
    }
```

在doJoin()方法里，首先通过查看任务的状态，看任务是否已经执行完了，如果执行完了，则直接返回任务状态，如果没有执行完，则从任务数组里取出任务并执行。如果任务顺利执行完成了，则设置任务状态为NORMAL，如果出现异常，则纪录异常，并将任务状态设置为EXCEPTIONAL。

7. 参考资料

- JDK1.7源码
- http://ifeve.com/fork-join-5/

8. 作者介绍

方腾飞，花名清英，并发编程网站站长。目前在阿里巴巴微贷事业部工作。并发编程网：http://ifeve.com，个人微博：http://weibo.com/kirals，欢迎通过我的微博进行技术交流。

感谢张龙对本文的审校。

给InfoQ中文站投稿或者参与内容翻译工作，请邮件至editors@cn.infoq.com。也欢迎大家通过新浪微博（@InfoQ）或者腾讯微博（@InfoQ）关注我们，并与我们的编辑和其他读者朋友交流。

相关内容

- 我们是如何优化HAProxy以让其支持2,000,000个并发SSL连接的？ 2017年6月14日

- 去哪儿网机票搜索系统的高并发架构设计 2017年4月21日

- 架构师（2017年10月） 2017年10月8日
- Gradle 4.2发布 2017年10月9日
- 与Brian Goetz聊Java的模式匹配 2017年10月9日
- Oracle完成OpenJDK 10软件仓库群的合并 2017年10月9日
- Spring 5.0 GA版本发布，支持JDK9及反应式编程 2017年9月30日
- IntelliJ IDEA宣布对Java 9的支持情况 2017年9月28日
- NetBeans第一部分代码提交Apache 2017年9月25日
- Kotlin与Java之争 2017年9月22日
- Oracle将Java EE移交Eclipse基金会 2017年9月22日

相关内容

- 期待已久的Java 9 今日发布 2017年9月21日

- Java平台组首席架构师谈Java的发展和迭代 2017年10月10日
- Eclipse更新了Eclipse公共许可（EPL） 2017年9月20日
- 美征信巨头Equifax因Struts漏洞导致数据大规模泄露 2017年9月18日
- Java EE Security API（JSR-375）获得通过 2017年9月18日
- Java社区对Java发布周期声明的反应 2017年9月18日
- Spring Boot 2.0将会增强Actuator端点的特性 2017年9月15日


- 架构师特刊：编程语言 2017年8月17日

- Java云托管服务的开支削减策略 2017年9月20日

- Cascade：自动化测试“旅程” 2017年9月19日

- Reladomo：自备全套功能的企业级开源Java ORM（二） 2017年8月24日

相关内容

- Reladomo：自备全套功能的企业级开源Java ORM（一） 2017年8月21日
- 案例学习：Jigsaw模块化迁移 2017年8月8日 

- 深入探索JVM自动资源管理 2017年7月11日 
- 架构师（2017年7月） 2017年7月8日 
- Java 老矣，尚能饭否？ 2017年6月29日 
- 想知道垃圾回收暂停的过程中发生了什么吗？查查垃圾回收日志就知道了！ 2017年5月17日 
- JVM上的确定性执行机制 2017年5月12日 
- 可编程世界的发展之路 2017年6月6日 
- Azure Stack设计哲学之物理架构探秘 2017年9月28日 
- 网易云原生架构实践之服务治理 2017年9月15日 
- 云深度学习平台架构与实践的必经之路 2017年9月13日 

InfoQ每周精要

订阅InfoQ每周精要，加入拥有25万多名资深开发者的庞大技术社区。

您的邮箱 订阅

语言 & 开发

W3C发布DRM作为推荐方案

新发布的CoffeeScript 2中添加了现代JavaScript特性

最新的.NET Framework聚焦于改进可访问性

架构 & 设计

从百度文件系统看大型分布式系统设计中的定式与创新

阿里新一代计算引擎Blink与SQL和机器学习的二三事

数据湖只是个哗众取宠的伪概念吗？

文化 & 方法

写博客如何助力敏捷团队

与Aurynn Shaw的问答：在DevOpsDays新西兰大会上分享个人DevOps经历

乐高的敏捷之旅

数据科学

Teachable Machine：训练机器在浏览器中使用摄像头

Apache Beam实战指南之基础入门

阿里达摩院成立，将解决1亿人口的就业问题

DevOps

与Aurynn Shaw的问答：在DevOpsDays新西兰大会上分享个人DevOps经历

Amazon新增对CloudWatch Dashboards Gains API和CloudFormation的支持

与Brian Goetz聊Java的模式匹配

- **信息无障碍参考文档**