链滴

Java 动手写爬虫 五 对象池

作者：YiHui

原文链接：https://ld246.com/article/1502027485424

来源网站：链滴

第五篇，对象池的设计与实现

前面每爬取一个任务都对应一个Job任务，试想一下，当我们爬取网页越来越多，速度越来越快时，会出现频繁的Job对象的创建和销毁，因此本片将考虑如何实现对象的复用，减少频繁的gc

设计

我们的目标是设计一个对象池，用于创建Job任务，基本要求是满足下面几点:

- 可以配置对象池的容量大小
- 通过对象池获取对象时，遵循一下规则：
    - 对象池中有对象时，总对象池中获取
    - 对象池中没有可用对象时，新创建对象返回（也可以采用阻塞，直到有可用对象，我们这里采直接创建新对象方式）
- 对象用完后扔回对象池

实现

1. 创建对象的工厂类 ObjectFactory

对象池在初始化对象时，借用对象工厂类来创建，实现解耦

```
public interface ObjectFactory<T> {

    T create();

}
```

2. 对象池中的对象接口 IPollCell

因为每个对象都拥有自己的作用域，内部包含一些成员变量，如果对象重用时，这些成员变量的值，能会造成影响，因此我们定义 IPoolCell 接口，其中声明一个方法，用于重置所有的变量信息

```
public interface IPoolCell {

    /**
     * 清空所有状态
     */
    void clear();

}
```

3. 一个简单的对象池 SimplePool

```
package com.quick.hui.crawler.core.pool;

import lombok.extern.slf4j.Slf4j;
```

```
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.atomic.AtomicInteger;

/**
 * Created by yihui on 2017/8/6.
 */
@Slf4j
public class SimplePool<T extends IPoolCell> {

    private static SimplePool instance;


    public static void initInstance(SimplePool simplePool) {
        instance = simplePool;
    }

    public static  SimplePool getInstance() {
        return instance;
    }


    private int size;

    private BlockingQueue<T> queue;

    private String name;

    private ObjectFactory objectFactory;


    private AtomicInteger objCreateCount = new AtomicInteger(0);

    public SimplePool(int size, ObjectFactory objectFactory) {
        this(size, "default-pool", objectFactory);
    }

    public SimplePool(int size, String name, ObjectFactory objectFactory) {
        this.size = size;
        this.name = name;
        this.objectFactory = objectFactory;

        queue = new LinkedBlockingQueue<>(size);
    }


    /**
     * 获取对象，若队列中存在， 则直接返回；若不存在，则新创建一个返回
     * @return
     */
    public T get() {
        T obj = queue.poll();
```

```
        if (obj != null) {
            return obj;
        }

        obj = (T) objectFactory.create();
        int num = objCreateCount.addAndGet(1);


        if (log.isDebugEnabled()) {
            if (objCreateCount.get() >= size) {
                log.debug("objectPoll fulled! create a new object! total create num: {}, poll size: {}",
um, queue.size());
            } else {
                // fixme 由于并发问题，这个队列的大小实际上与添加对象时的大小不一定相同
                log.debug("objectPoll not fulled!, init object, now poll size: {}", queue.size());
            }
        }

        return obj;
    }


    /**
     * 将对象扔回到队列中
     *
     * @param obj
     */
    public void release(T obj) {
        obj.clear();

        // 非阻塞方式的扔进队列
        boolean ans = queue.offer(obj);

        if (log.isDebugEnabled()) {
            log.debug("return obj to pool status: {}, now size: {}", ans, queue.size());
        }
    }


    public void clear() {
        queue.clear();
    }
}
```

上面的方法中，主要看get和release方法，简单说明

- get 方法
  - 首先是从队列中获取对象（非阻塞方式，获取不到时返回null而不是异常）
  - 队列为空时，新建一个对象返回
    - 未初始化队列，创建的对象表示可回收重复使用的
    - 队列填满了，但是被其他线程获取完了，此时创建的对象理论上不需要重复使用，用完一次丢掉

- release 方法
  - 清空对象状态
  - 扔进队列（非阻塞）

4. Job修改

既然要使用对象池，那么我们的IJob对象需要实现 IPoolCell接口了

将实现放在 DefaultAbstractCrawlJob 类中

```
@Override
public void clear() {
    this.depth = 0;
    this.crawlMeta = null;
    this.fetchQueue = null;
    this.crawlResult = null;
}
```

使用

上面只是实现了一个最简单的最基础的对象池，接下来就是适配我们的爬虫系统了

之前的创建Job任务是在 com.quick.hui.crawler.core.fetcher.Fetcher#start 中直接根据传入的class象来创建对象，因此，第一步就是着手改Fetcher类

1. 初始化对象池

创建方法修改，新增对象池对象初始化：Fetcher.java

```
public <T extends DefaultAbstractCrawlJob> Fetcher(Class<T> jobClz) {
    this(0, jobClz);
}


public <T extends DefaultAbstractCrawlJob> Fetcher(int maxDepth, Class<T> jobClz) {
    this(maxDepth, () -> {
        try {
            return jobClz.newInstance();
        } catch (Exception e) {
            log.error("create job error! e: {}", e);
            return null;
        }
    });
}

public <T extends DefaultAbstractCrawlJob> Fetcher(int maxDepth, ObjectFactory<T> jobFac
ory) {
    this.maxDepth = maxDepth;
    fetchQueue = FetchQueue.DEFAULT_INSTANCE;
    threadConf = ThreadConf.DEFAULT_CONF;
    initExecutor();
```

```
    SimplePool simplePool = new SimplePool<>(ConfigWrapper.getInstance().getConfig().getF
tchQueueSize(), jobFactory);
    SimplePool.initInstance(simplePool);
}
```

说明

为什么将创建的对象池座位 SimplePool的静态变量？

因为每个任务都是异步执行，在任务执行完之后扔回队列，这个过程不在 Fetcher对象中执行，为了享对象池，采用了这种猥琐的方法

2. 启动方法修改

在创建 Fetcher 对象时，已经初始化好对象池，因此start方法不需要接收参数，直接改为

```
public <T extends DefaultAbstractCrawlJob> void start() throws Exception {
  ....

  DefaultAbstractCrawlJob job = (DefaultAbstractCrawlJob) SimplePool.getInstance().get();
  job.setDepth(this.maxDepth);
  job.setCrawlMeta(crawlMeta);
  job.setFetchQueue(fetchQueue);

  executor.execute(job);

  ...
```

测试

测试代码与之前有一点区别，即 Fetcher 在创建时选择具体的Job对象类型，其他的没啥区别

```
public static class QueueCrawlerJob extends DefaultAbstractCrawlJob {

    public void beforeRun() {
        // 设置返回的网页编码
        super.setResponseCode("gbk");
    }

    @Override
    protected void visit(CrawlResult crawlResult) {
//            System.out.println(Thread.currentThread().getName() + "___" + crawlMeta.getCurren
Depth() + "___" + crawlResult.getUrl());
    }
}


@Test
public void testCrawel() throws Exception {
    Fetcher fetcher = new Fetcher(2, QueueCrawlerJob.class);

    String url = "http://chengyu.911cha.com/zishu_4.html";
```

```
    CrawlMeta crawlMeta = new CrawlMeta();
    crawlMeta.setUrl(url);
    crawlMeta.addPositiveRegex("http://chengyu.911cha.com/zishu_4_p[0-9]+\\.html$");

    fetcher.addFeed(crawlMeta);


    fetcher.start();
}
```

待改进

上面只是实现了一个最基本简单的对象池，有不少可以改进的地方

- 对象池实例的维护，上面是采用静态变量方式，局限太强，导致这个对象池无法多个共存
- 对象池大小没法动态配置，初始化时设置好了之后就没法改
- 可考虑新增阻塞方式的获取对象

以上坑留待后续有空进行修改

3. 源码地址

项目地址： https://github.com/liuyueyi/quick-crawler

对象池对应的tag: v0.008

参考

- 一个通用并发对象池的实现