OkHttp3源码分析[任务队列]


BlackSwift (/u/b99b0edd4e77) ＋关注
2016.01.28 13:59* 字数 1955 阅读 15220 评论 32 喜欢 44 赞赏 1
(/u/b99b0edd4e77)

OkHttp系列文章如下

- OkHttp3源码分析[综述] (https://www.jianshu.com/p/aad5aacd79bf)
- OkHttp3源码分析[复用连接池] (https://www.jianshu.com/p/92a61357164b)
- OkHttp3源码分析[缓存策略] (https://www.jianshu.com/p/9cebbbd0eeab)
- OkHttp3源码分析[DiskLruCache] (https://www.jianshu.com/p/23b8aa490a6b)
- OkHttp3源码分析[任务队列] (https://www.jianshu.com/p/6637369d02e7)

本文目录：

1. 线程池基础
2. 反向代理模型
3. OkHttp的任务调度

看过Wiki的都知道OkHttp拥有2种运行方式，一种是同步阻塞调用并直接返回的形式，另一种是通过内部线程池分发调度实现非阻塞的异步回调。本文主要分析第二种，即OkHttp在多并发网络下的分发调度过程。本文主要分析的是 `Dispatcher` 对象

---

1. 线程池基础

在初学Java的时候，各位可能会用 `new Thread + Handler` 来写异步任务，它的坑网上已经烂大街了，比如不能自动关闭，迷之缩进难以维护，导致目前开发者几乎不怎么用它。而现在很多框架，比如Picasso，Rxjava等，都帮我们写好了对应场景的线程池，但是线程池到底有什么好呢？

1.1. 线程池好处都有啥

线程池的关键在于线程复用以减少非核心任务的损耗。下面内容是引用IBM知识库(https://link.jianshu.com?t=https://www.ibm.com/developerworks/cn/java/l-threadPool/)中的例子：

> 多线程技术主要解决处理器单元内多个线程执行的问题，它可以显著减少处理器单元的闲置时间，增加处理器单元的吞吐能力。但如果对多线程应用不当，会增加对单个任务的处理时间。可以举一个简单的例子：
> 假设在一台服务器完成一项任务的时间为T

>

T1 创建线程的时间
T2 在线程中执行任务的时间，包括线程间同步所需时间
T3 线程销毁的时间

```
>显然T ＝ T1＋T2＋T3。注意这是一个极度简化的假设。
可以看出T1,T3是多线程本身的带来的开销（在Java中，通过映射pThead，并进一步通过SystemCall实现native线

池技术正是关注如何缩短或调整T1，T3时间的技术，从而提高服务器程序性能的。

1. 通过对线程进行缓存，减少了创建销毁的时间损失
2. 通过控制线程数量阀值，减少了当线程过少时带来的CPU闲置（比如说长时间卡在I/O上了）与线程过多时对JVM

>类似的还有Socket连接池、[DB连接池](https://github.com/alibaba/druid)、CommonPool(比如Jedis)等

在Java中，我们可以通过`线程池工厂`或者`自定义参数`来创建线程池，这些[教程](http://www.jianshu.com/

###1.2. OkHttp配置的线程池

在OkHttp，使用如下构造了单例线程池

```java
public synchronized ExecutorService executorService() {
  if (executorService == null) {
    executorService = new ThreadPoolExecutor(0, Integer.MAX_VALUE, 60, TimeUnit.SECONDS,
        new SynchronousQueue<Runnable>(), Util.threadFactory("OkHttp Dispatcher", false));
  }
  return executorService;
}
```

参数说明如下：

- int corePoolSize: 最小并发线程数，这里并发同时包括空闲与活动的线程，如果是0的话，空闲一段时间后所有线程将全部被销毁。
- int maximumPoolSize: 最大线程数，当任务进来时可以扩充的线程最大值，当大于了这个值就会根据丢弃处理机制来处理
- long keepAliveTime: 当线程数大于 `corePoolSize` 时，多余的空闲线程的最大存活时间，类似于HTTP中的Keep-alive
- TimeUnit unit: 时间单位，一般用秒
- BlockingQueue<Runnable> workQueue: 工作队列，先进先出，可以看出并不像Picasso那样设置优先队列。
- ThreadFactory threadFactory: 单个线程的工厂，可以打Log，设置 `Daemon` (即当JVM退出时，线程自动结束)等

可以看出，在Okhttp中，构建了一个阀值为[0, Integer.MAX_VALUE]的线程池，它不保留任何最小线程数，随时创建更多的线程数，当线程空闲时只能活60秒，它使用了一个不存储元素的阻塞工作队列，一个叫做"OkHttp Dispatcher"的线程工厂。

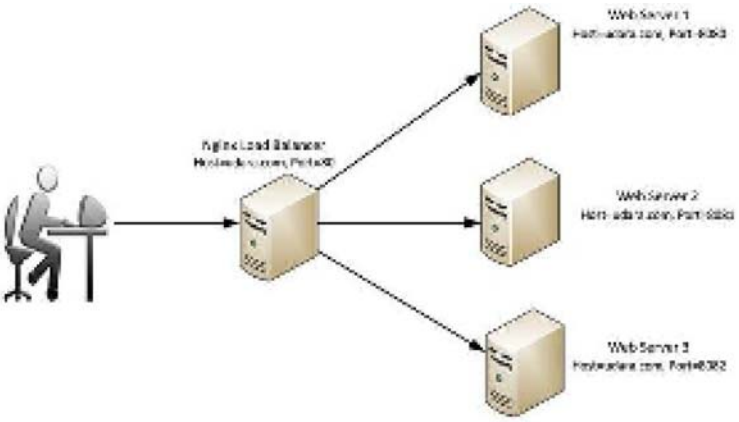
也就是说，在实际运行中，当收到10个并发请求时，线程池会创建十个线程，当工作完成后，线程池会在60s后相继关闭所有线程。

> 在RxJava的 `Schedulers.io()` 中，也有类似的设计，只不过是线程池的池，最小的线程数量控制，不设上限的最大线程，以保证I/O任务中高阻塞低占用的过程中，不会长时间卡在阻塞上，有兴趣的可以分析RxJava中4种不同场景的Schedulers

反向代理模型

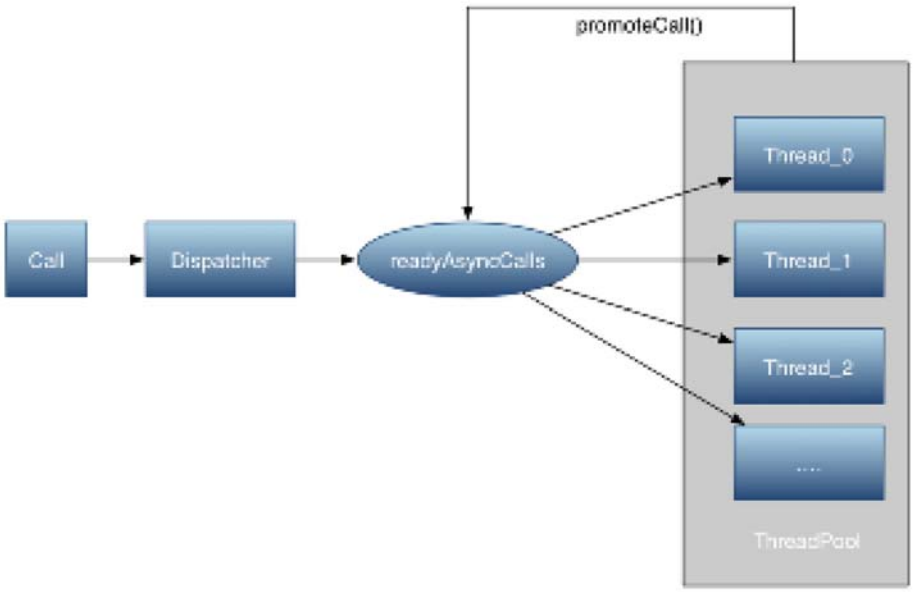
在OkHttp中，使用了与Nginx类似的反向代理与分发技术，这是典型的单生产者多消费者问题。

我们知道在Nginx/SLB中，用户通过HTTP(Socket)访问前置的服务器，服务器会添加Header并自动转发请求给后端集群，接着返回数据结果给用户(比如简书上次挂了也显示了Nginx报错)。通过将工作分配给多个后台服务器并共享Redis的Session，可以提高服务的负载均衡能力，实现**非阻塞、高可用、高并发连接**，避免资源全部放到一台服务器而带来的负载，速度，在线率等影响。
```



Nginx Load balancing

而在OkHttp中，非常类似于上述场景，它使用Dispatcher (https://link.jianshu.com?t=https://github.com/square/okhttp/blob/7826bcb2fb1facb697a4c512776756c05d8c9deb/okhttp/src/main/java/okhttp3/Dispatcher.java#L38-L38)作为任务的派发器，线程池对应多台后置服务器，用 `AsyncCall` 对应Socket请求，用 `Deque<readyAsyncCalls>` 对应Nginx的内部缓存



Okhttp Dispatcher

具体成员如下

- maxRequests = 64: 最大并发请求数为64
- maxRequestsPerHost = 5: 每个主机最大请求数为5
- Dispatcher: 分发者，也就是生产者（默认在主线程）
- AsyncCall: 队列中需要处理的Runnable（包装了异步回调接口）
- ExecutorService：消费者池（也就是线程池）
- Deque<readyAsyncCalls>：缓存（用数组实现，可自动扩容，无大小限制）
- Deque<runningAsyncCalls>：正在运行的任务，仅仅是用来引用正在运行的任务以判断并发量，注意它并不是消费者缓存

通过将请求任务分发给多个线程，可以显著的减少I/O等待时间

OkHttp的任务调度

当我们希望使用OkHttp的异步请求时，一般进行如下构造

```
OkHttpClient client = new OkHttpClient.Builder().build();
Request request = new Request.Builder()
    .url("http://qq.com").get().build();
client.newCall(request).enqueue(new Callback() {
  @Override public void onFailure(Call call, IOException e) {

  }

  @Override public void onResponse(Call call, Response response) throws IOException {

  }
});
```

当HttpClient的请求入队 (https://link.jianshu.com?t=https://github.com/square/okhttp/blob/7826bcb2fb1facb697a4c512776756c05d8c9deb/okhttp/src/main/java/okhttp3/Dispatcher.java#L110-L110)时，根据代码，我们可以发现实际上是Dispatcher进行了入队 (https://link.jianshu.com?t=https://github.com/square/okhttp/blob/7826bcb2fb1facb697a4c512776756c05d8c9deb/okhttp/src/main/java/okhttp3/Dispatcher.java#L109)操作

```
synchronized void enqueue(AsyncCall call) {
  if (runningAsyncCalls.size() < maxRequests && runningCallsForHost(call) < maxRequestsPerHc
    //添加正在运行的请求
    runningAsyncCalls.add(call);
    //线程池执行请求
    executorService().execute(call);
  } else {
    //添加到缓存队列排队等待
    readyAsyncCalls.add(call);
  }
}
```

可以发现请求是否进入缓存的条件如下：

```
(runningRequests<64 && runningRequestsPerHost<5)
```

如果满足条件，那么就直接把 `AsyncCall` 直接加到 `runningCalls` 的队列中，并在线程池中执行（线程池会根据当前负载自动创建，销毁，缓存相应的线程）。反之就放入 `readyAsyncCalls` 进行缓存等待。

我们再分析请求元素AsyncCall（它实现了Runnable接口），它内部实现 (https://link.jianshu.com?t=https://github.com/square/okhttp/blob/8ff37250310e8d2f9e73293199b3b6e42ec45b0f/okhttp/src/main/java/okhttp3/RealCall.java#L124)的execute方法如下

```
@Override protected void execute() {
  boolean signalledCallback = false;
  try {
    //执行耗时IO任务
    Response response = getResponseWithInterceptorChain(forWebSocket);
    if (canceled) {
      signalledCallback = true;
      //回调，注意这里回调是在线程池中，而不是想当然的主线程回调
      responseCallback.onFailure(RealCall.this, new IOException("Canceled"));
    } else {
      signalledCallback = true;
      //回调，同上
      responseCallback.onResponse(RealCall.this, response);
    }
  } catch (IOException e) {
    if (signalledCallback) {
      // Do not signal the callback twice!
      logger.log(Level.INFO, "Callback failure for " + toLoggableString(), e);
    } else {
      responseCallback.onFailure(RealCall.this, e);
    }
  } finally {
    //最关键的代码
    client.dispatcher().finished(this);
  }
}
```

当任务执行完成后，无论是否有异常，`finally`代码段**总会**被执行，也就是会调用Dispatcher的finished (https://link.jianshu.com?t=https://github.com/square/okhttp/blob/7826bcb2fb1facb697a4c512776756c05d8c9deb/okhttp/src/main/java/okhttp3/Dispatcher.java#L137)函数，打开源码，发现它将正在运行的任务Call从队列runningAsyncCalls中移除后，接着执行`promoteCalls()`函数

```
private void promoteCalls() {
    //如果目前是最大负荷运转，接着等
  if (runningAsyncCalls.size() >= maxRequests) return; // Already running max capacity.
  //如果缓存等待区是空的，接着等
  if (readyAsyncCalls.isEmpty()) return; // No ready calls to promote.

  for (Iterator<AsyncCall> i = readyAsyncCalls.iterator(); i.hasNext(); ) {
    AsyncCall call = i.next();

    if (runningCallsForHost(call) < maxRequestsPerHost) {
        //将缓存等待区最后一个移动到运行区中，并执行
      i.remove();
      runningAsyncCalls.add(call);
      executorService().execute(call);
    }

    if (runningAsyncCalls.size() >= maxRequests) return; // Reached max capacity.
  }
}
```

这样，就主动的把缓存队列向前走了一步，而没有使用互斥锁等复杂编码

Summary

通过上述的分析，我们知道了：

1. OkHttp采用Dispatcher技术，类似于Nginx，与线程池配合实现了高并发，低阻塞的运行
2. Okhttp采用Deque作为缓存，按照入队的顺序先进先出
3. OkHttp最出彩的地方就是在try/finally中调用了`finished`函数，可以主动控制等待队列的移动，而不是采用锁或者wait/notify，极大减少了编码复杂性

还有一句话，如果将OkHttp作为底层框架使用的话，我建议还是**使用同步方式**进行包装，并自己定制线程调度。比如将Quartz与OkHttp进行异步调用时，任务的执行就会错乱导致最后出现并发问题。

Refference