前面的博客已经提到过，OkHttp的一个高效之处在于在内部维护了一个线程池，方便高效地执行异步请求。本篇博客将详细介绍OkHttp的任务队列机制。

1. 线程池的优点

OkHttp的任务队列在内部维护了一个线程池用于执行具体的网络请求。而线程池最大的好处在于通过线程复用减少非核心任务的损耗。

> 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线程），我们渴望减少T1,T3所用的时间，从而减少T的时间。但一些线>程的使用者并没有注意到这一点，所以在程序中频繁的创建或销毁线程，这导致T1和T3在T中占有>相当比例。显然这是突出了线程的弱点（T1，T3），而不是优点（并发性）。

线程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的内存与线程切换时系统调用的压力

> 类似的还有Socket连接池、DB连接池 (https://github.com/alibaba/druid)、CommonPool(比如Jedis)等技术。

2. OkHttp的任务队列

OkHttp的任务队列主要由两部分组成：

- 任务分发器dispatcher：负责为任务找到合适的执行线程
- 网络请求任务线程池

```
public final class Dispatcher {
  private int maxRequests = 64;
  private int maxRequestsPerHost = 5;
  private Runnable idleCallback;

  /** Executes calls. Created lazily. */
  private ExecutorService executorService;

  /** Ready async calls in the order they'll be run. */
  private final Deque<AsyncCall> readyAsyncCalls = new ArrayDeque<>();

  /** Running asynchronous calls. Includes canceled calls that haven't finished yet. */
  private final Deque<AsyncCall> runningAsyncCalls = new ArrayDeque<>();

  /** Running synchronous calls. Includes canceled calls that haven't finished yet. */
  private final Deque<RealCall> runningSyncCalls = new ArrayDeque<>();

  public Dispatcher(ExecutorService executorService) {
    this.executorService = executorService;
  }

  public Dispatcher() {
  }

  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：

- readyAsyncCalls：待执行异步任务队列
- runningAsyncCalls：运行中异步任务队列
- runningSyncCalls：运行中同步任务队列
- executorService：任务队列线程池：

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory) {
    this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
         threadFactory, defaultHandler);
```

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 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

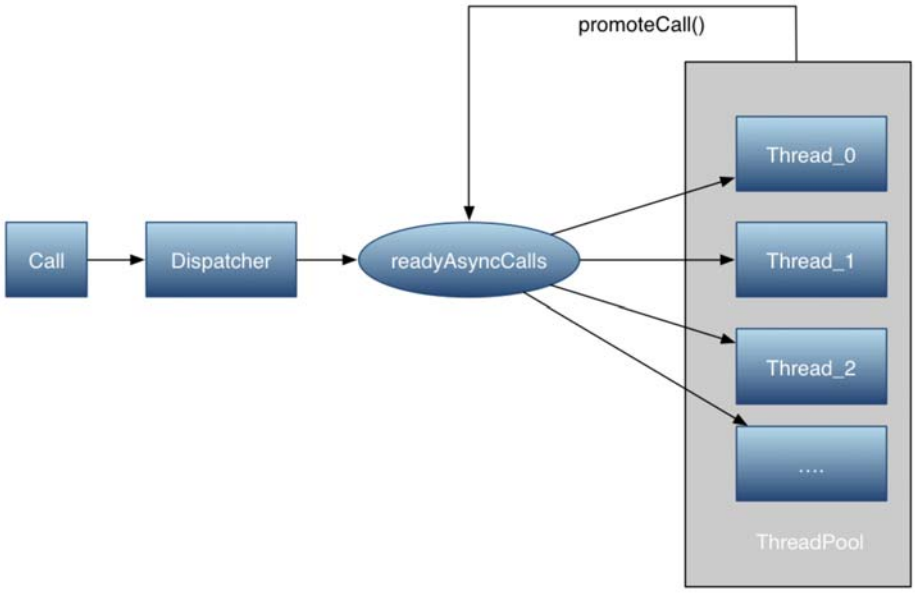
ThreadFactory threadFactory: 单个线程的工厂，可以打Log，设置 `Daemon` (即当JVM退出时，线程自动结束)等

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。

3. Dispatcher分发器

dispatcher分发器类似于Ngnix中的反向代理，通过Dispatcher将任务分发到合适的空闲线程，实现 非阻塞，高可用，高并发连接



1.同步请求

当我们使用OkHttp进行同步请求时，一般构造如下：

```
OkHttpClient client = new OkHttpClient();
Request request = new Request.Builder()
    .url("http://publicobject.com/helloworld.txt")
    .build();
Response response = client.newCall(request).execute();
```

接下来看看 `RealCall.execute`

```
  @Override public Response execute() throws IOException {
    synchronized (this) {
      if (executed) throw new IllegalStateException("Already Executed");
      executed = true;
    }
    captureCallStackTrace();
    try {
      client.dispatcher().executed(this);
      Response result = getResponseWithInterceptorChain();
      if (result == null) throw new IOException("Canceled");
      return result;
    } finally {
      client.dispatcher().finished(this);
    }
  }
```

同步调用的执行逻辑是：

- 将对应任务加入分发器
- 执行任务
- 执行完成后通知dispatcher对应任务已完成，对应任务出队

2.异步请求

异步请求一般构造如下：

```
        OkHttpClient client = new OkHttpClient();
        Request request = new Request.Builder()
                .url("http://publicobject.com/helloworld.txt")
                .build();

        client.newCall(request).enqueue(new Callback() {
            @Override
            public void onFailure(Call call, IOException e) {
                Log.d("OkHttp", "Call Failed:" + e.getMessage());
            }

            @Override
            public void onResponse(Call call, Response response) throws IOException {
                Log.d("OkHttp", "Call succeeded:" + response.message());
            }
        });
```

当HttpClient的请求入队时，根据代码，我们可以发现实际上是Dispatcher进行了入队操作。

```
synchronized void enqueue(AsyncCall call) {
  if (runningAsyncCalls.size() < maxRequests && runningCallsForHost(call) < maxRequestsPerH
     //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);
  }
}
```

如果满足条件：

- 当前请求数小于最大请求数（64）
- 对单一host的请求小于阈值（5）

将该任务插入正在执行任务队列，并执行对应任务。如果不满足则将其放入待执行队列。

接下来看看 AsyncCall.execute

```
@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，无论成功与否都会调用dispatcher.finished方法，通知分发器相关任务已结束：

```
  private <T> void finished(Deque<T> calls, T call, boolean promoteCalls) {
    int runningCallsCount;
    Runnable idleCallback;
    synchronized (this) {
      if (!calls.remove(call)) throw new AssertionError("Call wasn't in-flight!");
      if (promoteCalls) promoteCalls();
      runningCallsCount = runningCallsCount();
      idleCallback = this.idleCallback;
    }

    if (runningCallsCount == 0 && idleCallback != null) {
      idleCallback.run();
    }
  }
```

- 空闲出多余线程，调用promoteCalls调用待执行的任务
- 如果当前整个线程池都空闲下来，执行空闲通知回调线程(idleCallback)

接下来看看promoteCalls：

```
private void promoteCalls() {
  if (runningAsyncCalls.size() >= maxRequests) return; // Already running max capacity.
  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) {
      i.remove();
      runningAsyncCalls.add(call);
      executorService().execute(call);
    }

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

promoteCalls的逻辑也很简单：扫描待执行任务队列，将任务放入正在执行任务队列，并执行该任务。

4. 总结

以上就是整个任务队列的实现细节，总结起来有以下几个特点：

- 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
- Okhttp采用Deque作为缓存，按照入队的顺序先进先出
- 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