Dispatcher类，在Okhttp框架中，起到了调度器的作用。管理着请求队列，我们来看Dispatcher所持有的变量：

```
  private int maxRequests = 64;//1
  private int maxRequestsPerHost = 5;//2
  private @Nullable Runnable idleCallback;//3

  /** Executes calls. Created lazily. */
  private @Nullable ExecutorService executorService;//4

  /** Ready async calls in the order they'll be run. */
  private final Deque<AsyncCall> readyAsyncCalls = new ArrayDeque<>();//5

  /** Running asynchronous calls. Includes canceled calls that haven't finished
yet. */
  private final Deque<AsyncCall> runningAsyncCalls = new ArrayDeque<>
();//6

  /** Running synchronous calls. Includes canceled calls that haven't finished
yet. */
  private final Deque<RealCall> runningSyncCalls = new ArrayDeque<>();//7
```

1:最大并发请求数为64；

2:每个host最大请求数为5；

3:线程池为空时回调。Set a callback to be invoked each time the dispatcher becomes idle (when the number of running calls returns to zero).

4:线程池

```
public synchronized ExecutorService executorService() {
  if (executorService == null) {
    executorService = new ThreadPoolExecutor(①0,
②Integer.MAX_VALUE, ③60, ④TimeUnit.SECONDS,
        ⑤new SynchronousQueue<Runnable>(),
⑥Util.threadFactory("OkHttp Dispatcher", false));
  }
  return executorService;
}
```

①corePoolSize:核心线程数量，保持在线程池中的线程数量(即使已经空闲)，为0代表线程空闲后不会保留，等待一段时间后停止。

②maximumPoolSize:表示线程池可以容纳最大线程数量,这里Integer.MAX_VALUE

③和④：组合表示当线程池中的线程数量大于0(核心线程)时，空闲的线程就会等待60s才会被终止。如果小于，则会立刻停止

⑤线程等待队列。同步队列，按序排队，先来先服务

⑥Util.threadFactory("OkHttp Dispatcher", false):线程工厂，直接创建一个名为OkHttp Dispatcher的非守护线程。

5:预备队列，正准备准备消费的队列；

6:正在运行的异步请求队列；

7:正在运行的同步请求队列；

这里有两个数据结构，一个是ArrayDeque，一个是SynchronousQueue，这里不过多分析源码，主要了解这两种数据结构的概念和定义：

SynchronousQueue是同步队列接口BloueckingQueue的实现类

SynchronousQueue的特征是在队列里最多只能有一个元素。当一个线程往队列里插入元素时，此线程将被阻塞直到另一个线程从队列中取出元素。如果一个线程从空的队列里取出元素，此线程将被阻塞直到另一个线程往队列中插入元素。

ArrayDeque是双端队列Deque接口的实现类

ArrayDeque是一种具有队列和栈性质的抽象数据类型。双端队列中的元素可以从两端弹出，插入和删除操作限定在队列的两边进行。既能先进先出，也能先进后出，可以作为栈来使用，效率高于Stack；也可以作为队列来使用，效率高于LinkedList。

RealCall封装了每一次的请求，在执行请求时调用自身方法：

同步：

```
public Response execute() throws IOException {
   synchronized (this) {
//忽略code
.....
client.dispatcher().executed(this);
 Response result = getResponseWithInterceptorChain();
//忽略code
.....
   } finally {
    client.dispatcher().finished(this);
   }
 }
```

异步：

```
public void enqueue(Callback responseCallback) {
//忽略code
....
  client.dispatcher().enqueue(new AsyncCall(responseCallback));
 }
```

均是调用的dispatcher对象内方法。回到dispatcher类内部：

同步：

```
/** Used by {@code Call#execute} to signal it is in-flight. */
 synchronized void executed(RealCall call) {
```

```
    runningSyncCalls.add(call);
  }
```

finally：

```
  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();
    }
  }
```

同步请求是直接将RealCall放入runningSyncCalls,并在finished中将其remove「finally中代码一定会执行」，finish中三个参数分别代表:队列;请求;是否优化在等待队列中的calls到运行队列中(大概是这么个意思，分析promoteCalls方法时就会明白这个参数的意思)。
同步请求调用的finished的方法是

```
void finished(RealCall call) {    finished(runningSyncCalls, call, false);
  }
```

异步：

```
synchronized void enqueue(AsyncCall call) {
    if (runningAsyncCalls.size() < maxRequests && runningCallsForHost(call) < maxRequestsPerHost) {
      runningAsyncCalls.add(call);//1
      executorService().execute(call);//2
```

```
  } else {
    readyAsyncCalls.add(call);//3
  }
 }
```

1，2，3:异步请求正在运行的请求小于64，且这个call请求的host的请求数小于5的话，异步运行队列加入这个call，并用线程池去执行它。else，异步预备队列中加入这个call。finished什么时候执行呢，往下看

注:注意第二点的execute执行的是AsyncCall内的execute()方法。因为AsyncCall继承于NamedRunnable，NamedRunnable实现了Runnable接口，并在run方法内执行了execute();

```
protected void execute() {
     boolean signalledCallback = false;
     try {
      Response response = getResponseWithInterceptorChain();
     //忽略代码....
     } catch (IOException e) {
  //忽略代码...
     } finally {
       client.dispatcher().finished(this);
     }
   }
 }
```

在finally中执行了:

```
 void finished(AsyncCall call) {
   finished(runningAsyncCalls, call, true);
 }
```

是否优化在等待队列中的calls到运行队列中:

```
  private void promoteCalls() {
    if (runningAsyncCalls.size() >= maxRequests) return; // Already running
max capacity.
```

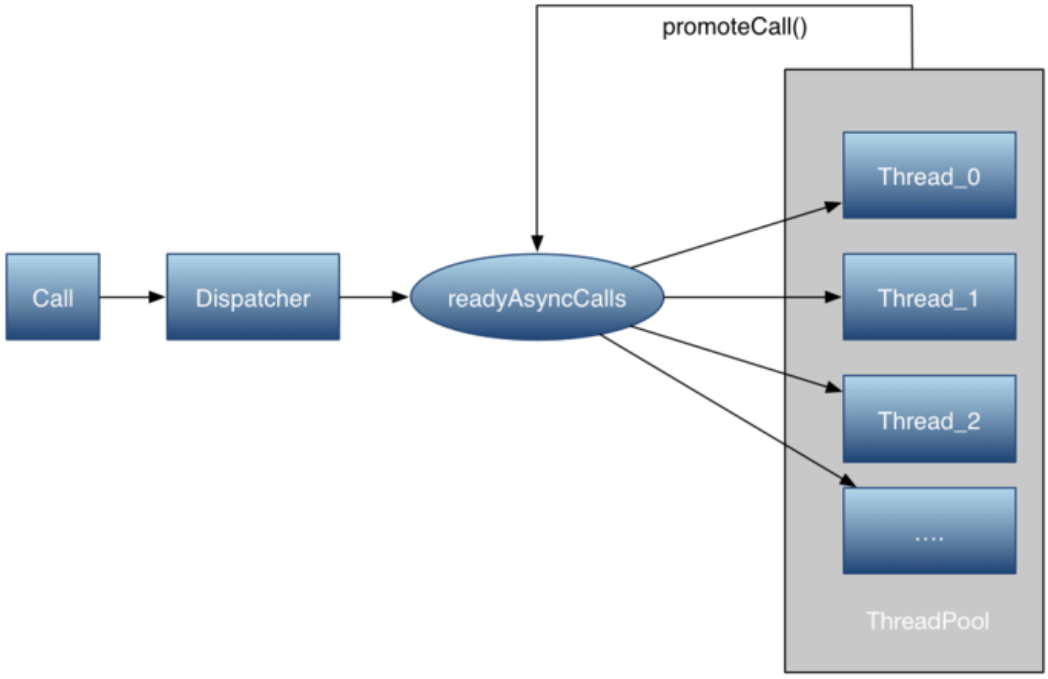
```
    if (readyAsyncCalls.isEmpty()) return; // No ready calls to promote.

    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.
    }
  }
```

正在运行的异步calls大于等于64，不优化了，return。
准备队列中没有calls，没什么需要优化，return。
开始遍历预备异步队列：如果这个host的请求数还少于5:
准备队列remove，线程池执行。
如果runningAsyncCalls大于64了，return。

```
/**
* 这段抄的
*/
```

OKHttp调度的优雅之处：
1、采用Dispacher作为调度，与线程池配合实现了高并发，低阻塞的的运行
2、采用Deque作为集合，按照入队的顺序先进先出
3、最精彩的就是在try/catch/finally中调用finished函数，可以主动控制队列的移动。避免了使用锁而wait/notify操作。

最后附带一张网图：


promoteCall()
Call
Dispatcher
readyAsyncCalls
Thread_0
Thread_1
Thread_2
....
ThreadPool