合理地利用本地缓存可以有效地减少网络开销，减少响应延迟。HTTP报头也定义了很多与缓存有关的域来控制缓存。今天就来讲讲OkHttp中关于缓存部分的实现细节。

1. HTTP缓存策略

首先来了解下HTTP协议中缓存部分的相关域。

1.1 Expires

超时时间，一般用在服务器的response报头中用于告知客户端对应资源的过期时间。当客户端需要再次请求相同资源时先比较其过期时间，如果尚未超过过期时间则直接返回缓存结果，如果已经超过则重新请求。

1.2 Cache-Control

相对值，单位时秒，表示当前资源的有效期。 `Cache-Control` 比 `Expires` 优先级更高：

```
Cache-Control:max-age=31536000,public
```

1.3 条件GET请求

1.3.1 Last-Modified-Date

客户端第一次请求时，服务器返回：

```
Last-Modified: Tue, 12 Jan 2016 09:31:27 GMT
```

当客户端二次请求时，可以头部加上如下header:

```
If-Modified-Since: Tue, 12 Jan 2016 09:31:27 GMT
```

如果当前资源没有被二次修改，服务器返回304告知客户端直接复用本地缓存。

1.3.2 ETag

ETag是对资源文件的一种摘要，可以通过ETag值来判断文件是否有修改。当客户端第一次请求某资源时，服务器返回：

```
ETag: "5694c7ef-24dc"
```

客户端再次请求时，可在头部加上如下域：

```
If-None-Match: "5694c7ef-24dc"
```

如果文件并未改变，则服务器返回304告知客户端可以复用本地缓存。

1.4 no-cache/no-store

不使用缓存

1.5 only-if-cached

只使用缓存

2. Cache源码分析

OkHttp的缓存工作都是在 `CacheInterceptor` 中完成的,Cache部分有如下几个关键类：

- Cache：Cache管理器，其内部包含一个DiskLruCache将cache写入文件系统:

```
 * <h3>Cache Optimization</h3>
 *
 * <p>To measure cache effectiveness, this class tracks three statistics:
 * <ul>
 *     <li><strong>{@linkplain #requestCount() Request Count:}</strong> the num
 *         requests issued since this cache was created.
 *     <li><strong>{@linkplain #networkCount() Network Count:}</strong> the num
 *         requests that required network use.
 *     <li><strong>{@linkplain #hitCount() Hit Count:}</strong> the number of t
 *         whose responses were served by the cache.
 * </ul>
 *
 * Sometimes a request will result in a conditional cache hit. If the cache cor
 * the response, the client will issue a conditional {@code GET}. The server wi
 * the updated response if it has changed, or a short 'not modified' response i
 * is still valid. Such responses increment both the network count and hit cour
 *
 * <p>The best way to improve the cache hit rate is by configuring the web serv
 * cacheable responses. Although this client honors all <a
 * href="http://tools.ietf.org/html/rfc7234">HTTP/1.1 (RFC 7234)</a> cache head
 * partial responses.
```

Cache内部通过 `requestCount` , `networkCount` , `hitCount` 三个统计指标来优化缓存效率

- CacheStrategy：缓存策略。其内部维护一个request和response，通过指定request和response来描述是通过网络还是缓存获取response，抑或二者同时使用

```
[CacheStrategy.java]
/**
* Given a request and cached response, this figures out whether to use the netw
* both.
*
* <p>Selecting a cache strategy may add conditions to the request (like the "If
* header for conditional GETs) or warnings to the cached response (if the cache
* potentially stale).
*/
public final class CacheStrategy {
 /** The request to send on the network, or null if this call doesn't use the r
 public final Request networkRequest;

 /** The cached response to return or validate; or null if this call doesn't us
 public final Response cacheResponse;
 ......
}
```

- CacheStrategy$Factory:缓存策略工厂类根据实际请求返回对应的缓存策略

既然实际的缓存工作都是在 `CacheInterceptor` 中完成的，那么接下来看下 `CahceInterceptor` 的核心方法 `intercept` 方法源码:

```
[CacheInterceptor.java]
@Override public Response intercept(Chain chain) throws IOException {
    //首先尝试获取缓存
    Response cacheCandidate = cache != null
        ? cache.get(chain.request())
        : null;

    long now = System.currentTimeMillis();

      //获取缓存策略
    CacheStrategy strategy = new CacheStrategy.Factory(now, chain.request(), cacheCandidate
    Request networkRequest = strategy.networkRequest;
    Response cacheResponse = strategy.cacheResponse;

    //如果有缓存，更新下相关统计指标：命中率
    if (cache != null) {
      cache.trackResponse(strategy);
```

```
}

//如果当前缓存不符合要求，将其close
if (cacheCandidate != null && cacheResponse == null) {
  closeQuietly(cacheCandidate.body()); // The cache candidate wasn't applicable. Close
}

// 如果不能使用网络，同时又没有符合条件的缓存，直接抛504错误
if (networkRequest == null && cacheResponse == null) {
  return new Response.Builder()
      .request(chain.request())
      .protocol(Protocol.HTTP_1_1)
      .code(504)
      .message("Unsatisfiable Request (only-if-cached)")
      .body(Util.EMPTY_RESPONSE)
      .sentRequestAtMillis(-1L)
      .receivedResponseAtMillis(System.currentTimeMillis())
      .build();
}

// 如果有缓存同时又不使用网络，则直接返回缓存结果
if (networkRequest == null) {
  return cacheResponse.newBuilder()
      .cacheResponse(stripBody(cacheResponse))
      .build();
}

//尝试通过网络获取回复
Response networkResponse = null;
try {
  networkResponse = chain.proceed(networkRequest);
} finally {
  // If we're crashing on I/O or otherwise, don't leak the cache body.
  if (networkResponse == null && cacheCandidate != null) {
    closeQuietly(cacheCandidate.body());
  }
}

// 如果既有缓存，同时又发起了请求，说明此时是一个Conditional Get请求
if (cacheResponse != null) {
  // 如果服务端返回的是NOT_MODIFIED,缓存有效，将本地缓存和网络响应做合并
  if (networkResponse.code() == HTTP_NOT_MODIFIED) {
    Response response = cacheResponse.newBuilder()
        .headers(combine(cacheResponse.headers(), networkResponse.headers()))
        .sentRequestAtMillis(networkResponse.sentRequestAtMillis())
        .receivedResponseAtMillis(networkResponse.receivedResponseAtMillis())
        .cacheResponse(stripBody(cacheResponse))
        .networkResponse(stripBody(networkResponse))
        .build();
    networkResponse.body().close();

    // Update the cache after combining headers but before stripping the
    // Content-Encoding header (as performed by initContentStream()).
    cache.trackConditionalCacheHit();
    cache.update(cacheResponse, response);
    return response;
  } else {// 如果响应资源有更新，关掉原有缓存
    closeQuietly(cacheResponse.body());
  }
}

Response response = networkResponse.newBuilder()
    .cacheResponse(stripBody(cacheResponse))
    .networkResponse(stripBody(networkResponse))
    .build();

if (cache != null) {
  if (HttpHeaders.hasBody(response) && CacheStrategy.isCacheable(response, networkReque
    // 将网络响应写入cache中
    CacheRequest cacheRequest = cache.put(response);
    return cacheWritingResponse(cacheRequest, response);
  }

  if (HttpMethod.invalidatesCache(networkRequest.method())) {
    try {
      cache.remove(networkRequest);
    } catch (IOException ignored) {
      // The cache cannot be written.
```

```
        }
      }
    }

    return response;
  }
```

核心逻辑都以中文注释的形式在代码中标注出来了，大家看代码即可。通过上面的代码可以看出，几乎所有的动作都是以CacheStrategy缓存策略为依据做出的，那么接下来看下缓存策略是如何生成的，相关代码实现在 `CacheStrategy$Factory.get()` 方法中：

```
[CacheStrategy$Factory]

    /**
     * Returns a strategy to satisfy {@code request} using the a cached response {@code res
     */
    public CacheStrategy get() {
      CacheStrategy candidate = getCandidate();

      if (candidate.networkRequest != null && request.cacheControl().onlyIfCached()) {
        // We're forbidden from using the network and the cache is insufficient.
        return new CacheStrategy(null, null);
      }

      return candidate;
    }

    /** Returns a strategy to use assuming the request can use the network. */
    private CacheStrategy getCandidate() {
      // 若本地没有缓存，发起网络请求
      if (cacheResponse == null) {
        return new CacheStrategy(request, null);
      }

      // 如果当前请求是HTTPS，而缓存没有TLS握手，重新发起网络请求
      if (request.isHttps() && cacheResponse.handshake() == null) {
        return new CacheStrategy(request, null);
      }

      // If this response shouldn't have been stored, it should never be used
      // as a response source. This check should be redundant as long as the
      // persistence store is well-behaved and the rules are constant.
      if (!isCacheable(cacheResponse, request)) {
        return new CacheStrategy(request, null);
      }


      //如果当前的缓存策略是不缓存或者是conditional get，发起网络请求
      CacheControl requestCaching = request.cacheControl();
      if (requestCaching.noCache() || hasConditions(request)) {
        return new CacheStrategy(request, null);
      }

      //ageMillis:缓存age
      long ageMillis = cacheResponseAge();
      //freshMillis：缓存保鲜时间
      long freshMillis = computeFreshnessLifetime();

      if (requestCaching.maxAgeSeconds() != -1) {
        freshMillis = Math.min(freshMillis, SECONDS.toMillis(requestCaching.maxAgeSeconds()
      }

      long minFreshMillis = 0;
      if (requestCaching.minFreshSeconds() != -1) {
        minFreshMillis = SECONDS.toMillis(requestCaching.minFreshSeconds());
      }

      long maxStaleMillis = 0;
      CacheControl responseCaching = cacheResponse.cacheControl();
      if (!responseCaching.mustRevalidate() && requestCaching.maxStaleSeconds() != -1) {
        maxStaleMillis = SECONDS.toMillis(requestCaching.maxStaleSeconds());
      }

      //如果 age + min-fresh >= max-age && age + min-fresh < max-age + max-stale，则虽然缓存
    if (!responseCaching.noCache() && ageMillis + minFreshMillis < freshMillis + maxStale
      Response.Builder builder = cacheResponse.newBuilder();
      if (ageMillis + minFreshMillis >= freshMillis) {
        builder.addHeader("Warning", "110 HttpURLConnection \"Response is stale\"");
      }
      long oneDayMillis = 24 * 60 * 60 * 1000L;
      if (ageMillis > oneDayMillis && isFreshnessLifetimeHeuristic()) {
        builder.addHeader("Warning", "113 HttpURLConnection \"Heuristic expiration\"");
      }
      return new CacheStrategy(null, builder.build());
    }

    // 发起conditional get请求
    String conditionName;
    String conditionValue;
    if (etag != null) {
      conditionName = "If-None-Match";
      conditionValue = etag;
    } else if (lastModified != null) {
      conditionName = "If-Modified-Since";
      conditionValue = lastModifiedString;
    } else if (servedDate != null) {
      conditionName = "If-Modified-Since";
      conditionValue = servedDateString;
    } else {
      return new CacheStrategy(request, null); // No condition! Make a regular request.
    }

    Headers.Builder conditionalRequestHeaders = request.headers().newBuilder();
    Internal.instance.addLenient(conditionalRequestHeaders, conditionName, conditionValue

    Request conditionalRequest = request.newBuilder()
        .headers(conditionalRequestHeaders.build())
        .build();
    return new CacheStrategy(conditionalRequest, cacheResponse);
  }
```

可以看到其核心逻辑在getCandidate函数中。基本就是HTTP缓存协议的实现，核心代码逻辑已通过中文注释说明，大家直接看代码就好。

3. DiskLruCache

Cache内部通过DiskLruCache管理cache在文件系统层面的创建，读取，清理等等工作，接下来看下DiskLruCache的主要逻辑：

```
public final class DiskLruCache implements Closeable, Flushable {

  final FileSystem fileSystem;
  final File directory;
  private final File journalFile;
  private final File journalFileTmp;
  private final File journalFileBackup;
  private final int appVersion;
  private long maxSize;
  final int valueCount;
  private long size = 0;
  BufferedSink journalWriter;
  final LinkedHashMap<String, Entry> lruEntries = new LinkedHashMap<>(0, 0.75f, true);

  // Must be read and written when synchronized on 'this'.
  boolean initialized;
  boolean closed;
  boolean mostRecentTrimFailed;
  boolean mostRecentRebuildFailed;

  /**
   * To differentiate between old and current snapshots, each entry is given a sequence num
   * time an edit is committed. A snapshot is stale if its sequence number is not equal to
   * entry's sequence number.
   */
  private long nextSequenceNumber = 0;

  /** Used to run 'cleanupRunnable' for journal rebuilds. */
  private final Executor executor;
  private final Runnable cleanupRunnable = new Runnable() {
    public void run() {
        ......
    }
  };
  ...
  }
```

3.1 journalFile

DiskLruCache内部日志文件，对cache的每一次读写都对应一条日志记录，DiskLruCache通过分析日志分析和创建cache。日志文件格式如下：

```
      libcore.io.DiskLruCache
      1
      100
      2

      CLEAN 3400330d1dfc7f3f7f4b8d4d803dfcf6 832 21054
      DIRTY 335c4c6028171cfddfbaae1a9c313c52
      CLEAN 335c4c6028171cfddfbaae1a9c313c52 3934 2342
      REMOVE 335c4c6028171cfddfbaae1a9c313c52
      DIRTY 1ab96a171faeeee38496d8b330771a7a
      CLEAN 1ab96a171faeeee38496d8b330771a7a 1600 234
      READ 335c4c6028171cfddfbaae1a9c313c52
      READ 3400330d1dfc7f3f7f4b8d4d803dfcf6

     前5行固定不变，分别为：常量：libcore.io.DiskLruCache；diskCache版本；应用程序版本；valueCou

     接下来每一行对应一个cache entry的一次状态记录，其格式为：[状态（DIRTY,CLEAN,READ,REMOVE），
     - DIRTY:表明一个cache entry正在被创建或更新，每一个成功的DIRTY记录都应该对应一个CLEAN或REMOV
     - CLEAN:说明cache已经被成功操作，当前可以被正常读取。每一个CLEAN行还需要记录其每一个value的长
     - READ: 记录一次cache读取操作
     - REMOVE:记录一次cache清除
```

日志文件的应用场景主要有四个：

- DiskCacheLru初始化时通过读取日志文件创建cache容器：lruEntries。同时通过日志过滤操作不成功的cache项。相关逻辑在DiskLruCache.readJournalLine,DiskLruCache.processJournal
- 初始化完成后，为避免日志文件不断膨胀，对日志进行重建精简，具体逻辑在DiskLruCache.reb

uildJournal

- 每当有cache操作时将其记录入日志文件中以备下次初始化时使用
- 当冗余日志过多时，通过调用cleanUpRunnable线程重建日志

3.2 DiskLruCache.Entry

每一个DiskLruCache.Entry对应一个cache记录：

```
private final class Entry {
  final String key;

  /** Lengths of this entry's files. */
  final long[] lengths;
  final File[] cleanFiles;
  final File[] dirtyFiles;

  /** True if this entry has ever been published. */
  boolean readable;

  /** The ongoing edit or null if this entry is not being edited. */
  Editor currentEditor;

  /** The sequence number of the most recently committed edit to this entry. */
  long sequenceNumber;

  Entry(String key) {
    this.key = key;

    lengths = new long[valueCount];
    cleanFiles = new File[valueCount];
    dirtyFiles = new File[valueCount];

    // The names are repetitive so re-use the same builder to avoid allocations.
    StringBuilder fileBuilder = new StringBuilder(key).append('.');
    int truncateTo = fileBuilder.length();
    for (int i = 0; i < valueCount; i++) {
      fileBuilder.append(i);
      cleanFiles[i] = new File(directory, fileBuilder.toString());
      fileBuilder.append(".tmp");
      dirtyFiles[i] = new File(directory, fileBuilder.toString());
      fileBuilder.setLength(truncateTo);
    }
  }
  ...

      /**
   * Returns a snapshot of this entry. This opens all streams eagerly to guarantee that we
   * single published snapshot. If we opened streams lazily then the streams could come fr
   * different edits.
   */
  Snapshot snapshot() {
    if (!Thread.holdsLock(DiskLruCache.this)) throw new AssertionError();

    Source[] sources = new Source[valueCount];
    long[] lengths = this.lengths.clone(); // Defensive copy since these can be zeroed out
    try {
      for (int i = 0; i < valueCount; i++) {
        sources[i] = fileSystem.source(cleanFiles[i]);
      }
      return new Snapshot(key, sequenceNumber, sources, lengths);
    } catch (FileNotFoundException e) {
      // A file must have been deleted manually!
      for (int i = 0; i < valueCount; i++) {
        if (sources[i] != null) {
          Util.closeQuietly(sources[i]);
        } else {
          break;
        }
      }
      // Since the entry is no longer valid, remove it so the metadata is accurate (i.e. t
      // size.)
      try {
        removeEntry(this);
      } catch (IOException ignored) {
      }
      return null;
    }
  }
}
```

一个Entry主要由以下几部分构成：

- key：每个cache都有一个key作为其标识符。当前cache的key为其对应URL的MD5字符串

- cleanFiles/dirtyFiles：每一个Entry对应多个文件，其对应的文件数由DiskLruCache.valueCount指定。当前在OkHttp中valueCount为2。即每个cache对应2个cleanFiles，2个dirtyFiles。其中第一个cleanFiles/dirtyFiles记录cache的meta数据（如URL,创建时间，SSL握手记录等等），第二个文件记录cache的真正内容。cleanFiles记录处于稳定状态的cache结果，dirtyFiles记录处于创建或更新状态的cache
- currentEditor：entry编辑器，对entry的所有操作都是通过其编辑器完成。编辑器内部添加了同步锁

3.3 cleanupRunnable

清理线程，用于重建精简日志：

```
private final Runnable cleanupRunnable = new Runnable() {
  public void run() {
    synchronized (DiskLruCache.this) {
      if (!initialized || closed) {
        return; // Nothing to do
      }

      try {
        trimToSize();
      } catch (IOException ignored) {
        mostRecentTrimFailed = true;
      }

      try {
        if (journalRebuildRequired()) {
          rebuildJournal();
          redundantOpCount = 0;
        }
      } catch (IOException e) {
        mostRecentRebuildFailed = true;
        journalWriter = Okio.buffer(Okio.blackhole());
      }
    }
  }
};
```

其触发条件在journalRebuildRequired()方法中：

```
/**
 * We only rebuild the journal when it will halve the size of the journal and eliminate a
 * 2000 ops.
 */
boolean journalRebuildRequired() {
  final int redundantOpCompactThreshold = 2000;
  return redundantOpCount >= redundantOpCompactThreshold
      && redundantOpCount >= lruEntries.size();
}
```

当冗余日志超过日志文件本身的一般且总条数超过2000时执行

3.4 SnapShot

cache快照，记录了特定cache在某一个特定时刻的内容。每次向DiskLruCache请求时返回的都是目标cache的一个快照,相关逻辑在DiskLruCache.get中：

```
[DiskLruCache.java]
/**
   * Returns a snapshot of the entry named {@code key}, or null if it doesn't exist is not
   * readable. If a value is returned, it is moved to the head of the LRU queue.
   */
  public synchronized Snapshot get(String key) throws IOException {
    initialize();

    checkNotClosed();
    validateKey(key);
    Entry entry = lruEntries.get(key);
    if (entry == null || !entry.readable) return null;

    Snapshot snapshot = entry.snapshot();
    if (snapshot == null) return null;

    redundantOpCount++;
    //日志记录
    journalWriter.writeUtf8(READ).writeByte(' ').writeUtf8(key).writeByte('\n');
    if (journalRebuildRequired()) {
      executor.execute(cleanupRunnable);
    }

    return snapshot;
  }
```

3.5 lruEntries

管理cache entry的容器，其数据结构是LinkedHashMap。通过LinkedHashMap本身的实现逻辑达到cache的LRU替换

3.6 FileSystem

使用Okio对File的封装，简化了I/O操作。

3.7 DiskLruCache.edit

DiskLruCache可以看成是Cache在文件系统层的具体实现，所以其基本操作接口存在一一对应的关系：

- Cache.get() —>DiskLruCache.get()
- Cache.put()—>DiskLruCache.edit() //cache插入
- Cache.remove()—>DiskLruCache.remove()
- Cache.update()—>DiskLruCache.edit()//cache更新

其中get操作在3.4已经介绍了，remove操作较为简单，put和update大致逻辑相似，因为篇幅限制，这里仅介绍Cache.put操作的逻辑，其他的操作大家看代码就好:

```
[okhttp3.Cache.java]
  CacheRequest put(Response response) {
    String requestMethod = response.request().method();

    if (HttpMethod.invalidatesCache(response.request().method())) {
      try {
        remove(response.request());
      } catch (IOException ignored) {
        // The cache cannot be written.
      }
      return null;
    }
    if (!requestMethod.equals("GET")) {
      // Don't cache non-GET responses. We're technically allowed to cache
      // HEAD requests and some POST requests, but the complexity of doing
      // so is high and the benefit is low.
      return null;
    }

    if (HttpHeaders.hasVaryAll(response)) {
      return null;
    }

    Entry entry = new Entry(response);
    DiskLruCache.Editor editor = null;
    try {
      editor = cache.edit(key(response.request().url()));
      if (editor == null) {
        return null;
      }
      entry.writeTo(editor);
      return new CacheRequestImpl(editor);
    } catch (IOException e) {
      abortQuietly(editor);
      return null;
    }
  }
```

可以看到核心逻辑在 `editor = cache.edit(key(response.request().url()));` ,相关代码在DiskLruCache.edit:

```
[okhttp3.internal.cache.DiskLruCache.java]
synchronized Editor edit(String key, long expectedSequenceNumber) throws IOException {
    initialize();

    checkNotClosed();
    validateKey(key);
    Entry entry = lruEntries.get(key);
    if (expectedSequenceNumber != ANY_SEQUENCE_NUMBER && (entry == null
        || entry.sequenceNumber != expectedSequenceNumber)) {
      return null; // Snapshot is stale.
    }
    if (entry != null && entry.currentEditor != null) {
      return null; // 当前cache entry正在被其他对象操作
    }
    if (mostRecentTrimFailed || mostRecentRebuildFailed) {
      // The OS has become our enemy! If the trim job failed, it means we are storing more
      // requested by the user. Do not allow edits so we do not go over that limit any furth
      // the journal rebuild failed, the journal writer will not be active, meaning we will
      // able to record the edit, causing file leaks. In both cases, we want to retry the cl
      // so we can get out of this state!
      executor.execute(cleanupRunnable);
      return null;
    }

    // 日志接入DIRTY记录
    journalWriter.writeUtf8(DIRTY).writeByte(' ').writeUtf8(key).writeByte('\n');
    journalWriter.flush();

    if (hasJournalErrors) {
      return null; // Don't edit; the journal can't be written.
    }

    if (entry == null) {
      entry = new Entry(key);
      lruEntries.put(key, entry);
    }
    Editor editor = new Editor(entry);
    entry.currentEditor = editor;
    return editor;
  }
```

edit方法返回对应CacheEntry的editor编辑器。接下来再来看下 `Cache.put()` 方法的 `entry.writeTo(editor);` ,其相关逻辑：

```
[okhttp3.internal.cache.DiskLruCache.java]
public void writeTo(DiskLruCache.Editor editor) throws IOException {
      BufferedSink sink = Okio.buffer(editor.newSink(ENTRY_METADATA));

      sink.writeUtf8(url)
          .writeByte('\n');
      sink.writeUtf8(requestMethod)
          .writeByte('\n');
      sink.writeDecimalLong(varyHeaders.size())
          .writeByte('\n');
      for (int i = 0, size = varyHeaders.size(); i < size; i++) {
        sink.writeUtf8(varyHeaders.name(i))
            .writeUtf8(": ")
            .writeUtf8(varyHeaders.value(i))
            .writeByte('\n');
      }

      sink.writeUtf8(new StatusLine(protocol, code, message).toString())
          .writeByte('\n');
      sink.writeDecimalLong(responseHeaders.size() + 2)
          .writeByte('\n');
      for (int i = 0, size = responseHeaders.size(); i < size; i++) {
        sink.writeUtf8(responseHeaders.name(i))
            .writeUtf8(": ")
            .writeUtf8(responseHeaders.value(i))
            .writeByte('\n');
      }
      sink.writeUtf8(SENT_MILLIS)
          .writeUtf8(": ")
          .writeDecimalLong(sentRequestMillis)
          .writeByte('\n');
      sink.writeUtf8(RECEIVED_MILLIS)
          .writeUtf8(": ")
          .writeDecimalLong(receivedResponseMillis)
          .writeByte('\n');

      if (isHttps()) {
        sink.writeByte('\n');
        sink.writeUtf8(handshake.cipherSuite().javaName())
            .writeByte('\n');
        writeCertList(sink, handshake.peerCertificates());
        writeCertList(sink, handshake.localCertificates());
        // The handshake’s TLS version is null on HttpsURLConnection and on older cached re
        if (handshake.tlsVersion() != null) {
          sink.writeUtf8(handshake.tlsVersion().javaName())
              .writeByte('\n');
        }
      }
      sink.close();
    }
```

其主要逻辑就是将对应请求的meta数据写入对应CacheEntry的索引为ENTRY_METADATA（0）的dirtyfile中。

最后再来看 `Cache.put()` 方法的 `return new CacheRequestImpl(editor);`：

```
[okhttp3.Cache$CacheRequestImpl]
private final class CacheRequestImpl implements CacheRequest {
    private final DiskLruCache.Editor editor;
    private Sink cacheOut;
    private Sink body;
    boolean done;

    public CacheRequestImpl(final DiskLruCache.Editor editor) {
      this.editor = editor;
      this.cacheOut = editor.newSink(ENTRY_BODY);
      this.body = new ForwardingSink(cacheOut) {
        @Override public void close() throws IOException {
          synchronized (Cache.this) {
            if (done) {
              return;
            }
            done = true;
            writeSuccessCount++;
          }
          super.close();
          editor.commit();
        }
      };
    }

    @Override public void abort() {
      synchronized (Cache.this) {
        if (done) {
          return;
        }
        done = true;
        writeAbortCount++;
      }
      Util.closeQuietly(cacheOut);
      try {
        editor.abort();
      } catch (IOException ignored) {
      }
    }

    @Override public Sink body() {
      return body;
    }
  }
```

其中 `close` , `abort` 方法会调用 `editor.abort` 和 `editor.commit` 来更新日志， `editor.commit` 还会将dirtyFile重置为cleanFile作为稳定可用的缓存，相关逻辑在 `okhttp3.internal.cache.DiskLruCache$Editor.completeEdit` 中:

```
[okhttp3.internal.cache.DiskLruCache$Editor.completeEdit]
synchronized void completeEdit(Editor editor, boolean success) throws IOException {
    Entry entry = editor.entry;
    if (entry.currentEditor != editor) {
      throw new IllegalStateException();
    }

    // If this edit is creating the entry for the first time, every index must have a value
    if (success && !entry.readable) {
      for (int i = 0; i < valueCount; i++) {
        if (!editor.written[i]) {
          editor.abort();
          throw new IllegalStateException("Newly created entry didn't create value for inde
        }
        if (!fileSystem.exists(entry.dirtyFiles[i])) {
          editor.abort();
          return;
        }
      }
    }

    for (int i = 0; i < valueCount; i++) {
      File dirty = entry.dirtyFiles[i];
      if (success) {
        if (fileSystem.exists(dirty)) {
          File clean = entry.cleanFiles[i];
          fileSystem.rename(dirty, clean);//将dirtyfile置为cleanfile
          long oldLength = entry.lengths[i];
          long newLength = fileSystem.size(clean);
          entry.lengths[i] = newLength;
          size = size - oldLength + newLength;
        }
      } else {
        fileSystem.delete(dirty);//若失败则删除dirtyfile
      }
    }

    redundantOpCount++;
    entry.currentEditor = null;
    //更新日志
    if (entry.readable | success) {
      entry.readable = true;
      journalWriter.writeUtf8(CLEAN).writeByte(' ');
      journalWriter.writeUtf8(entry.key);
      entry.writeLengths(journalWriter);
      journalWriter.writeByte('\n');
      if (success) {
        entry.sequenceNumber = nextSequenceNumber++;
      }
    } else {
      lruEntries.remove(entry.key);
      journalWriter.writeUtf8(REMOVE).writeByte(' ');
      journalWriter.writeUtf8(entry.key);
      journalWriter.writeByte('\n');
    }
    journalWriter.flush();

    if (size > maxSize || journalRebuildRequired()) {
      executor.execute(cleanupRunnable);
    }
  }
```

CacheRequestImpl实现CacheRequest接口，向外部类(主要是CacheInterceptor)透出，外部对象通过CacheRequestImpl更新或写入缓存数据。

3.8总结

总结起来DiskLruCache主要有以下几个特点：

- 通过LinkedHashMap实现LRU替换
- 通过本地维护Cache操作日志保证Cache原子性与可用性，同时为防止日志过分膨胀定时执行日志精简

- 每一个Cache项对应两个状态副本：DIRTY,CLEAN。CLEAN表示当前可用状态Cache，外部访问到的cache快照均为CLEAN状态；DIRTY为更新态Cache。由于更新和创建都只操作DIRTY状态副本，实现了Cache的读写分离
- 每一个Cache项有四个文件，两个状态（DIRTY,CLEAN）,每个状态对应两个文件：一个文件存储Cache meta数据，一个文件存储Cache内容数据