

## 第四部分

# 案例研究

本书的最后一部分介绍的是 5 家知名公司使用 Netty 实现的任务关键型的系统的案例研究。第 14 章是关于 Droplr、Firebase 和 Urban Airship 的项目。第 15 章讨论了在 Facebook 和 Twitter 所完成的工作。

这些项目所描述的范围从核心的基础架构组件到移动服务以及新的网络协议，同时还包括了两个用于执行远程过程调用（RPC）的项目。在所有的这些案例中，你都将会看到这些组织已经通过 Netty 实现了你在本书中学到的相同的性能以及架构方面的优势。

# 第 14 章 案例研究，第一部分

## 本章主要内容

- Droplr
- Firebase
- Urban Airship

在本章中，我们将介绍两部分案例研究中的第一部分，它们是由已经在内部基础设施中广泛使用了 Netty 的公司贡献的。我们希望这些其他人如何利用 Netty 框架来解决现实世界问题的例子，能够拓展你对于 Netty 能够做到什么事情的理解。

**注意** 每个案例分析的作者都直接参与了他们所讨论的项目。

## 14.1 Droplr——构建移动服务

**Bruno de Carvalho，首席架构师**

在 Droplr，我们在我们的基础设施的核心部分、从我们的 API 服务器到辅助服务的各个部分都使用了 Netty。

这是一个关于我们是如何从一个单片的、运行缓慢的LAMP<sup>①</sup>应用程序迁移到基于Netty实现的现代的、高性能的以及水平扩展的分布式架构的案例研究。

### 14.1.1 这一切的起因

当我加入这个团队时，我们运行的是一个 LAMP 应用程序，其作为前端页面服务于用户，同时还作为 API 服务于客户端应用程序，其中，也包括我的逆向工程的、第三方的 Windows 客户端 windroplr。

① 一个典型的应用程序技术栈的首字母缩写；由 Linux、Apache Web Server、MySQL 以及 PHP 的首字母组成。

后来 Windroplr 变成了 Droplr for Windows，而我则开始主要负责基础设施的建设，并且最终得到了一个新的挑战：完全重新考虑 Droplr 的基础设施。

在那时，Droplr 本身已经确立成为了一种工作的理念，因此 2.0 版本的目标也是相当的标准：

- 将单片的技术栈拆分为多个可横向扩展的组件；
- 添加冗余，以避免宕机；
- 为客户端创建一个简洁的 API；
- 使其全部运行在 HTTPS 上。

创始人 Josh 和 Levi 对我说：“要不惜一切代价，让它飞起来。”

我知道这句话意味的可不只是变快一点或者变快很多。“要不惜一切代价”意味着一个完全数量级上的更快。而且我也知道，Netty 最终将会在这样的努力中发挥重要作用。

### 14.1.2 Droplr 是怎样工作的

Droplr 拥有一个非常简单的工作流：将一个文件拖动到应用程序的菜单栏图标，然后 Droplr 将会上传该文件。当上传完成之后，Droplr 将复制一个短 URL——也就是所谓的拖乐（drop）——到剪贴板。

就是这样。欢畅地、实时地分享。

而在幕后，拖乐元数据将会被存储到数据库中（包括创建日期、名称以及下载次数等信息），而文件本身则被存储在 Amazon S3 上。

### 14.1.3 创造一个更加快速的上传体验

Droplr 的第一个版本的上传流程是相当地天真可爱：

- (1) 接收上传；
- (2) 上传到 S3；
- (3) 如果是图片，则创建略缩图；
- (4) 应答客户端应用程序。

更加仔细地看看这个流程，你很快便会发现在第 2 步和第 3 步上有两个瓶颈。不管从客户端上传到我们的服务器有多快，在实际的上传完成之后，直到成功地接收到响应之间，对于拖乐的创建总是会有恼人的间隔——因为对应的文件仍然需要被上传到 S3 中，并为其生成略缩图。

文件越大，间隔的时间也越长。对于非常大的文件来说，连接<sup>①</sup>最终将会在等待来自服务器的响应时超时。由于这个严重的问题，当时Droplr只可以提供单个文件最大 32MB 的上传能力。

有两种截然不同的方案来减少上传时间。

- 方案 A，乐观且看似更加简单（见图 14-1）：

---

<sup>①</sup> 指客户端和服务器之间的连接。——译者注

- ◆ 完整地接收文件；
- ◆ 将文件保存到本地的文件系统，并立即返回成功到客户端；
- ◆ 计划在将来的某个时间点将其上传到 S3。
- 方案 B，安全但复杂（见图 14-2）：
  - ◆ 实时地（流式地）将从客户端上传的数据直接管道给 S3。

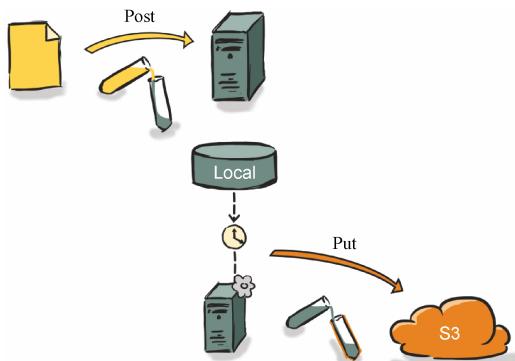


图 14-1 方案 A，乐观且看似更加简单

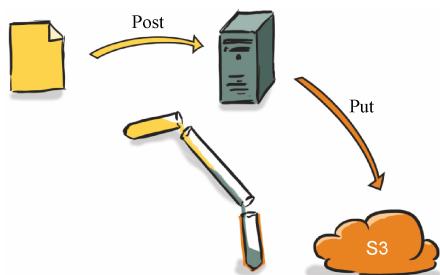


图 14-2 方案 B，安全但复杂

## 1. 乐观且看似更加简单的方案

在收到文件之后便返回一个短 URL 创造了一个空想（也可以将其称为隐式的契约），即该文件立即在该 URL 地址上可用。但是并不能够保证，上传的第二阶段（实际将文件推送到 S3）也将最终会成功，那么用户可能会得到一个坏掉的链接，其可能已经被张贴到了 Twitter 或者发送给了一个重要的客户。这是不可接受的，即使是每十万次上传也只会发生一次。

我们当前的数据显示，我们的上传失败率略低于 0.01%（万分之一），绝大多数都是在上传实际完成之前，客户端和服务器之间的连接就超时了。

我们也可以尝试通过在文件被最终推送到 S3 之前，从接收它的机器提供该文件的服务来绕开它，然而这种做法本身就是一堆麻烦：

- 如果在一批文件被完整地上传到 S3 之前，机器出现了故障，那么这些文件将会永久丢失；
- 也将会有跨集群的同步问题（“这个拖乐所对应的文件在哪里呢？”）；
- 将会需要额外的复杂的逻辑来处理各种边界情况，继而不断产生更多的边界情况；

在思考过每种变通方案和其陷阱之后，我很快认识到，这是一个经典的九头蛇问题——对于每个砍下的头，它的位置上都会再长出两个头。

## 2. 安全但复杂的方案

另一个选项需要对整体过程进行底层的控制。从本质上说，我们必须要能够做到以下几点。

- 在接收客户端上传文件的同时，打开一个到 S3 的连接。

- 将从客户端连接上收到的数据管道给到 S3 的连接。
- 缓冲并节流这两个连接：
  - ◆ 需要进行缓冲，以在客户端到服务器，以及服务器到 S3 这两个分支之间保持一条的稳定的流；
  - ◆ 需要进行节流，以防止当服务器到 S3 的分支上的速度变得慢于客户端到服务器的分支时，内存被消耗殆尽。
- 当出现错误时，需要能够在两端进行彻底的回滚。

看起来概念上很简单，但是它并不是你的通常的 Web 服务器能够提供的能力。尤其是当你考虑节流一个 TCP 连接时，你需要对它的套接字进行底层的访问。

它同时也引入了一个新的挑战，其将最终塑造我们的终极架构：推迟略缩图的创建。

这也意味着，无论该平台最终构建于哪种技术栈之上，它都必须要不仅能够提供一些基本的特性，如难以置信的性能和稳定性，而且在必要时还要能够提供操作底层（即字节级别的控制）的灵活性。

#### 14.1.4 技术栈

当开始一个新的 Web 服务器项目时，最终你将会问自己：“好吧，这些酷小子们这段时间都在用什么框架呢？”我也是这样的。

选择 Netty 并不是一件无需动脑的事；我研究了大量的框架，并谨记我认为的 3 个至关重要的要素。

(1) 它必须是快速的。我可不打算用一个低性能的技术栈替换另一个低性能的技术栈。  
(2) 它必须能够伸缩。不管它是有 1 个连接还是 10 000 个连接，每个服务器实例都必须要能够保持吞吐量，并且随着时间推移不能出现崩溃或者内存泄露。

(3) 它必须提供对底层数据的控制。字节级别的读取、TCP 拥塞控制等，这些都是难点。

要素 1 和要素 2 基本上排除了任何非编译型的语言。我是 Ruby 语言的拥趸，并且热爱 Sinatra 和 Padrino 这样的轻量级框架，但是我知道我所追寻的性能是不可能通过这些构件块实现的。

要素 2 本身就意味着：无论是什么样的解决方案，它都不能依赖于阻塞 I/O。看到了本书这里，你肯定已经明白为什么非阻塞 I/O 是唯一的选择了。

要素 3 比较绕弯儿。它意味着必须要在一个框架中找到完美的平衡，它必须在提供了对于它所接收到的数据的底层控制的同时，也支持快速的开发，并且值得信赖。这便是语言、文档、社区以及其他的成功案例开始起作用的时候了。

在那时我有一种强烈的感觉：Netty 便是我的首选武器。

##### 1. 基本要素：服务器和流水线

服务器基本上只是一个 ServerBootstrap，其内置了 NioServerSocketChannelFactory，配置了几个常见的 ChannelHandler 以及在末尾的 HTTP RequestController，如代码清单 14-1 所示。

## 代码清单 14-1 设置 ChannelPipeline

```

pipelineFactory = new ChannelPipelineFactory() {
    @Override
    public ChannelPipeline getPipeline() throws Exception {
        ChannelPipeline pipeline = Channels.pipeline();
        pipeline.addLast("idleStateHandler", new IdleStateHandler(...));
        pipeline.addLast("httpServerCodec", new HttpServerCodec());
        pipeline.addLast("requestController", new RequestController(...));
        return pipeline;
    }
};

IdleStateHandler 将
关闭不活动的连接
HttpServerCodec 将传入的字节
转换为 HttpRequest，并将传出
的 HttpResponse 转换为字节
将 RequestController
添加到 ChannelPipeline 中

```

RequestController 是 ChannelPipeline 中唯一自定义的 Droplr 代码，同时也可能是整个 Web 服务器中最复杂的部分。它的作用是处理初始请求的验证，并且如果一切都没问题，那么将会把请求路由到适当的请求处理器。对于每个已经建立的客户端连接，都会创建一个新的实例，并且只要连接保持活动就一直存在。

请求控制器负责：

- 处理负载洪峰；
- HTTP ChannelPipeline 的管理；
- 设置请求处理的上下文；
- 派生新的请求处理器；
- 向请求处理器供给数据；
- 处理内部和外部的错误。

代码清单 14-2 给出的是 RequestController 相关部分的一个纲要。

## 代码清单 14-2 RequestController

```

public class RequestController
    extends IdleStateAwareChannelUpstreamHandler {

    @Override
    public void channelIdle(ChannelHandlerContext ctx,
                           IdleStateEvent e) throws Exception {
        // Shut down connection to client and roll everything back.
    }

    @Override
    public void channelConnected(ChannelHandlerContext ctx,
                               ChannelStateEvent e) throws Exception {
        if (!acquireConnectionSlot()) {
            // Maximum number of allowed server connections reached,
            // respond with 503 service unavailable
            // and shutdown connection.
        } else {
            // Set up the connection's request pipeline.
        }
    }
}

```

```

    }

    @Override public void messageReceived(ChannelHandlerContext ctx,
        MessageEvent e) throws Exception {
        if (isDone()) return;
        if (e.getMessage() instanceof HttpRequest) {
            handleHttpRequest((HttpRequest) e.getMessage());
        } else if (e.getMessage() instanceof HttpChunk) {
            handleHttpChunk((HttpChunk)e.getMessage());
        }
    }
}

```

Droplr 的服务器请求验证的关键点

如果针对当前请求有一个活动的处理器，并且它能够接受 HttpChunk 数据，那么它将继续按 HttpChunk 传递

如同本书之前所解释过的一样，你应该永远不要在 Netty 的 I/O 线程上执行任何非 CPU 限定的代码——你将会从 Netty 偷取宝贵的资源，并因此影响到服务器的吞吐量。

因此，`HttpRequest` 和 `HttpChunk` 都可以通过切换到另一个不同的线程，来将执行流程移交给请求处理器。当请求处理器不是 CPU 限时，就会发生这样的情况，不管是因为它们访问了数据库，还是执行了不适合于本地内存或者 CPU 的逻辑。

当发生线程切换时，所有的代码块都必须要以串行的方式执行；否则，我们就会冒风险，对于一次上传来说，在处理完了序列号为  $n$  的 `HttpChunk` 之后，再处理序列号为  $n-1$  的 `HttpChunk` 必然会导致文件内容的损坏。（我们可能会交错所上传的文件的字节布局。）为了处理这种情况，我创建了一个自定义的线程池执行器，其确保了所有共享了同一个通用标识符的任务都将以串行的方式被执行。

从这里开始，这些数据（请求和 `HttpChunk`）便开始了在 Netty 和 Droplr 王国之外的冒险。

我将简短地解释请求处理器是如何被构建的，以在 `RequestController`（其存在于 Netty 的领地）和这些处理器（存在于 Droplr 的领地）之间的桥梁上亮起一些光芒。谁知道呢，这也许将会帮助你架构你自己的服务器应用程序呢！

## 2. 请求处理器

请求处理器提供了 Droplr 的功能。它们是类似地址为 `/account` 或者 `/drops` 这样的 URI 背后的端点。它们是逻辑核心——服务器对于客户端请求的解释器。

请求处理器的实现也是（Netty）框架实际上成为了 Droplr 的 API 服务器的地方。

## 3. 父接口

每个请求处理器，不管是直接的还是通过子类继承，都是 `RequestHandler` 接口的实现。

其本质上，`RequestHandler` 接口表示了一个对于请求（`HttpRequest` 的实例）和分块（`HttpChunk` 的实例）的无状态处理器。它是一个非常简单的接口，包含了一组方法以帮助请求控制器来执行以及/或者决定如何执行它的职责，例如：

- 请求处理器是有状态的还是无状态的呢？它需要从某个原型克隆，还是原型本身就可以用来处理请求呢？

- 请求处理器是 CPU 限定的还是非 CPU 限定的呢？它可以在 Netty 的工作线程上执行，还是需要在一个单独的线程池中执行呢？
- 回滚当前的变更；
- 清理任何使用过的资源。

这个接口<sup>①</sup>就是RequestController对于相关动作的所有理解。通过它非常清晰和简洁的接口，该控制器可以和有状态的和无状态的、CPU限定的和非CPU限定的（或者这些性质的组合）处理器以一种独立的并且实现无关的方式进行交互。

#### 4. 处理器的实现

最简单的 RequestHandler 实现是 AbstractRequestHandler，它代表一个子类型的层次结构的根，在到达提供了所有 Droplr 的功能的实际处理器之前，它将变得愈发具体。最终，它会到达有状态的实现 SimpleHandler，它在一个非 I/O 工作线程中执行，因此也不是 CPU 限定的。SimpleHandler 是快速实现那些执行读取 JSON 格式的数据、访问数据库，然后写出一些 JSON 的典型任务的端点的理想选择。

#### 5. 上传请求处理器

上传请求处理器是整个 Droplr API 服务器的关键。它是对于重塑 webserver 模块——服务器的框架化部分的设计的响应，也是到目前为止整个技术栈中最复杂、最优化的代码部分。

在上传的过程中，服务器具有双重行为：

- 在一边，它充当了正在上传文件的 API 客户端的服务器；
- 在另一边，它充当了 S3 的客户端，以推送它从 API 客户端接收的数据。

为了充当客户端，服务器使用了一个同样使用Netty构建的HTTP客户端库<sup>②③</sup>。这个异步的 HTTP 客户端库暴露了一组完美匹配该服务器的需求的接口。它将开始执行一个HTTP请求，并允许在数据变得可用时再供给给它，而这大大地降低了上传请求处理器的客户门面的复杂性。

##### 14.1.5 性能

在服务器的初始版本完成之后，我运行了一批性能测试。结果简直就是让人兴奋不已。在不断地增加了难以置信的负载之后，我看到新的服务器的上传在峰值时相比于旧版本的 LAMP 技术栈的快了 10 ~ 12 倍（完全数量级的更快），而且它能够支撑超过 1000 倍的并发上传，总共将近 10k 的并发上传（而这一切都只是运行在一个单一的 EC2 大型实例之上）。

下面的这些因素促成了这一点。

---

① 指 RequestHandler。——译者注

② 你可以在 <https://github.com/brunodecarvalho/http-client> 找到这个 HTTP 客户端库。

③ 上一个脚注中提到的这个 HTTP 客户端库已经废弃，推荐 AsyncHttpClient (<https://github.com/AsyncHttpClient/async-http-client>) 和 Akka-HTTP (<https://github.com/akka/akka-http>)，它们都实现了相同的功能。——译者注

- 它运行在一个调优的 JVM 中。
- 它运行在一个高度调优的自定义技术栈中，是专为解决这个问题而创建的，而不是一个通用的 Web 框架。
- 该自定义的技术栈通过 Netty 使用了 NIO（基于选择器的模型）构建，这意味着不同于每个客户端一个进程的 LAMP 技术栈，它可以扩展到上万甚至是几十万的并发连接。
- 再也没有以两个单独的，先接收一个完整的文件，然后再将其上传到 S3，的步骤所带来的开销了。现在文件将直接流向 S3。
- 因为服务器现在对文件进行了流式处理，所以：
  - ◆ 它再也不会花时间在 I/O 操作上了，即将数据写入临时文件，并在稍后的第二阶段上传中读取它们；
  - ◆ 对于每个上传也将消耗更少的内存，这意味着可以进行更多的并行上传。
- 略缩图生成变成了一个异步的后处理。

#### 14.1.6 小结——站在巨人的肩膀上

所有的这一切能够成为可能，都得益于 Netty 的难以置信的精心设计的 API，以及高性能的非阻塞的 I/O 架构。

自 2011 年 12 月推出 Droplr 2.0 以来，我们在 API 级别的宕机时间几乎为零。在几个月前，由于一次既定的全栈升级（数据库、操作系统、主要的服务器和守护进程的代码库升级），我们中断了已经连续一年半安静运行的基础设施的 100% 正常运行时间，这次升级只耗费了不到 1 小时的时间。

这些服务器日复一日地坚挺着，每秒钟处理几百个（有时甚至是几千个）并发请求，而同时还保持了如此低的内存和 CPU 使有率，以至于我们都难以相信它们实际上正在真实地做着如此大量的工作：

- CPU 使用率很少超过 5%；
- 无法准确地描述内存使用率，因为进程启动时预分配了 1 GB 的内存，同时配置的 JVM 可以在必要时增长到 2 GB，而在过去的两年内这一次也没有发生过。

任何人都可以通过增加机器来解决某个特定的问题，然而 Netty 帮助了 Droplr 智能地伸缩，并且保持了相当低的服务器账单。

## 14.2 Firebase——实时的数据同步服务

**Sara Robinson, Developer Happiness 副总裁**

**Greg Soltis, Cloud Architecture 副总裁**

实时更新是现代应用程序中用户体验的一个组成部分。随着用户期望这样的行为，越来越多的应用程序都正在实时地向用户推送数据的变化。通过传统的 3 层架构很难实现实时的数据同步，其需要开发者管理他们自己的运维、服务器以及伸缩。通过维护到客户端的实时的、双向的

通信，Firebase 提供了一种即时的直观体验，允许开发人员在几分钟之内跨越不同的客户端进行应用程序数据的同步——这一切都不需要任何的后端工作、服务器、运维或者伸缩。

实现这种能力提出了一项艰难的技术挑战，而 Netty 则是用于在 Firebase 内构建用于所有网络通信的底层框架的最佳解决方案。这个案例研究概述了 Firebase 的架构，然后审查了 Firebase 使用 Netty 以支撑它的实时数据同步服务的 3 种方式：

- 长轮询；
- HTTP 1.1 keep-alive 和流水线化；
- 控制 SSL 处理器。

### 14.2.1 Firebase 的架构

Firebase 允许开发者使用两层体系结构来上线运行应用程序。开发者只需要简单地导入 Firebase 库，并编写客户端代码。数据将以 JSON 格式暴露给开发者的代码，并且在本地进行缓存。该库处理了本地高速缓存和存储在 Firebase 服务器上的主副本（master copy）之间的同步。对于任何数据进行的更改都将会被实时地同步到与 Firebase 相连接的潜在的数十万个客户端上。跨多个平台的多个客户端之间的以及设备和 Firebase 之间的交互如图 14-3 所示。

Firebase 的服务器接收传入的数据更新，并将它们立即同步给所有注册了对于更改的数据感兴趣的已经连接的客户端。为了启用状态更改的实时通知，客户端将会始终保持一个到 Firebase 的活动连接。该连接的范围是：从基于单个 Netty Channel 的抽象到基于多个 Channel 的抽象，甚至是在客户端正在切换传输类型时的多个并存的抽象。

因为客户端可以通过多种方式连接到 Firebase，所以保持连接代码的模块化很重要。Netty 的 Channel 抽象对于 Firebase 集成新的传输来说简直是梦幻般的构建块。此外，流水线和处理器<sup>①</sup>模式使得可以简单地把传输相关的细节隔离开来，并为应用程序代码提供一个公共的消息流抽象。同样，这也极大地简化了添加新的协议支持所需要的工作。Firebase 只通过简单地添加几个新的 ChannelHandler 到 ChannelPipeline 中，便添加了对一种二进制传输的支持。对于实现客户端和服务器之间的实时连接而言，Netty 的速度、抽象的级别以及细粒度的控制都使得它成为了一个卓绝的框架。

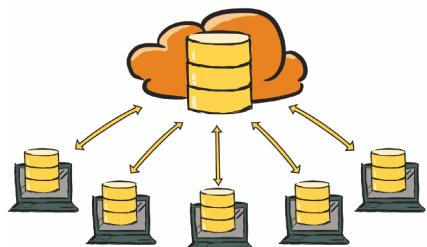


图 14-3 Firebase 的架构

### 14.2.2 长轮询

Firebase 同时使用了长轮询和 WebSocket 传输。长轮询传输是高度可靠的，覆盖了所有的浏览器、网络以及运营商；而基于 WebSocket 的传输，速度更快，但是由于浏览器/客户端的局限性，并不总是可用的。开始时，Firebase 将会使用长轮询进行连接，然后在 WebSocket 可用时再

① 指 ChannelPipeline 和 ChannelHandler。——译者注

升级到 WebSocket。对于少数不支持 WebSocket 的 Firebase 流量，Firebase 使用 Netty 实现了一个自定义的库来进行长轮询，并且经过调优具有非常高的性能和响应性。

Firebase 的客户端库逻辑处理双向消息流，并且会在任意一端关闭流时进行通知。虽然这在 TCP 或者 WebSocket 协议上实现起来相对简单，但是在处理长轮询传输时它仍然是一项挑战。对于长轮询的场景来说，下面两个属性必须被严格地保证：

- 保证消息的按顺序投递；
- 关闭通知。

### 1. 保证消息的按顺序投递

可以通过使得在某个指定的时刻有且只有一个未完成的请求，来实现长轮询的按顺序投递。因为客户端不会在它收到它的上一个请求的响应之前发出另一个请求，所以这就保证了它之前所发出的所有消息都被接收，并且可以安全地发送更多的请求了。同样，在服务器端，直到客户端收到之前的响应之前，将不会发出新的请求。因此，总是可以安全地发送缓存在两个请求之间的任何东西。然而，这将导致一个严重的缺陷。使用单一请求技术，客户端和服务器端都将花费大量的时间来对消息进行缓冲。例如，如果客户端有新的数据需要发送，但是这时已经有了一个未完成的请求，那么它在发出新请求之前，就必须得等待服务器的响应。如果这时在服务器上没有可用的数据，则可能需要很长的时间。

一个更加高性能的解决方案则是容忍更多的正在并发进行的请求。在实践中，这可以通过将单一请求的模式切换为最多两个请求的模式。这个算法包含了两个部分：

- 每当客户端有新的数据需要发送时，它都会发送一个新的请求，除非已经有了两个请求正在被处理；
- 每当服务器接收到来自客户端的请求时，如果它已经有了一个来自客户端的未完成的请求，那么即使没有数据，它也将立即回应第一个请求。

相对于单一请求的模式，这种方式提供了一个重要的改进：客户端和服务器的缓冲时间都被限定在了最多一次的网络往返时间里。

当然，这种性能的增加并不是没有代价的；它导致了代码复杂性的相应增加。该长轮询算法也不再保证消息的按顺序投递，但是一些来自 TCP 协议的理念可以保证这些消息的按顺序投递。由客户端发送的每个请求都包含一个序列号，每次请求时都将会递增。此外，每个请求都包含了关于有效负载中的消息数量的元数据。如果一个消息跨越了多个请求，那么在有效负载中所包含的消息的序号也会被包含在元数据中。

服务器维护了一个传入消息分段的环形缓冲区，在它们完成之后，如果它们之前没有不完整的消息，那么会立即对它们进行处理。下行要简单点，因为长轮询传输响应的是 HTTP GET 请求，而且对于有效载荷的大小没有相同的限制。在这种情况下，将包含一个对于每个响应都将会递增的序列号。只要客户端接收到了达到指定序列号的所有响应，它就可以开始处理列表中的所有消息；如果它还没有收到，那么它将缓冲该列表，直到它接收到了这些未完成的响应。

## 2. 关闭通知

在长轮询传输中第二个需要保证的属性是关闭通知。在这种情况下，使得服务器意识到传输已经关闭，明显要重要于使得客户端识别到传输的关闭。客户端所使用的 Firebase 库将会在连接断开时将操作放入队列以便稍后执行，而且这些被放入队列的操作可能也会对其他仍然连接着的客户端造成影响。因此，知道客户端什么时候实际上已经断开了是非常重要的。实现由服务器发起的关闭操作是相对简单的，其可以通过使用一个特殊的协议级别的关闭消息响应下一个请求来实现。

实现客户端的关闭通知是比较棘手的。虽然可以使用相同关闭通知，但是有两种情况可能会导致这种方式失效：用户可以关闭浏览器标签页，或者网络连接也可能会消失。标签页关闭的这种情况可以通过 `iframe` 来处理，`iframe` 会在页面卸载时发送一个包含关闭消息的请求。第二种情况则可以通过服务器端超时来处理。小心谨慎地选择超时值大小很重要，因为服务器无法区分慢速的网络和断开的客户端。也就是说，对于服务器来说，无法知道一个请求是被实际推迟了一分钟，还是该客户端丢失了它的网络连接。相对于应用程序需要多快地意识到断开的客户端来说，选取一个平衡了误报所带来的成本（关闭慢速网络上的客户端的传输）的合适的超时大小是很重要的。

图 14-4 演示了 Firebase 的长轮询传输是如何处理不同类型请求的。

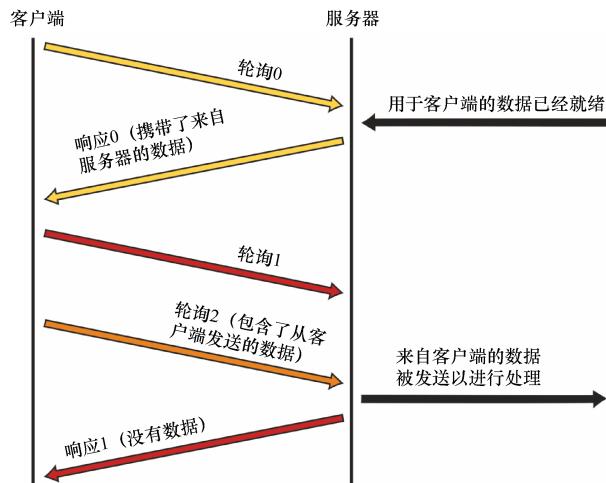


图 14-4 长轮询

在这个图中，每个长轮询请求都代表了不同类型的场景。最初，客户端向服务器发送了一个轮询（轮询 0）。一段时间之后，服务器从系统内的其他地方接收到了发送给该客户端的数据，所以它使用该数据响应了轮询 0。在该轮询返回之后，因为客户端目前没有任何未完成的请求，所以客户端又立即发送了一个新的轮询（轮询 1）。过了一小会儿，客户端需要发送数据给服务器。因为它只有一个未完成的轮询，所以它又发送了一个新的轮询（轮询 2），其中包含了需要

被递交的数据。根据协议, 一旦在服务器同时存在两个来自相同的客户端的轮询时, 它将响应第一个轮询。在这种情况下, 服务器没有任何已经就绪的数据可以用于该客户端, 因此它发送回了一个空响应。客户端也维护了一个超时, 并将在超时被触发时发送第二次轮询, 即使它没有任何额外的数据需要发送。这将系统从由于浏览器超时缓慢的请求所导致的故障中隔离开来。

### 14.2.3 HTTP 1.1 keep-alive 和流水线化

通过 HTTP 1.1 keep-alive 特性, 可以在同一个连接上发送多个请求到服务器。这使得 HTTP 流水线化——可以发送新的请求而不必等待来自服务器的响应, 成为了可能。实现对于 HTTP 流水线化以及 keep-alive 特性的支持通常是直截了当的, 但是当混入了长轮询之后, 它就明显变得更加复杂起来。

如果一个长轮询请求紧跟着一个 REST( 表征状态转移 ) 请求, 那么将有一些注意事项需要被考虑在内, 以确保浏览器能够正确工作。一个 Channel 可能会混和异步消息 ( 长轮询请求 ) 和同步消息 ( REST 请求 )。当一个 Channel 上出现了一个同步请求时, Firebase 必须按顺序同步响应应该 Channel 中所有之前的请求。例如, 如果有一个未完成的长轮询请求, 那么在处理该 REST 请求之前, 需要使用一个空操作对该长轮询传输进行响应。

图 14-5 说明了 Netty 是如何让 Firebase 在一个套接字上响应多个请求的。

如果浏览器有多个打开的连接, 并且正在使用长轮询, 那么它将重用这些连接来处理来自这两个打开的标签页的消息。对于长轮询请求来说, 这是很困难的, 并且还需要妥善地管理一个 HTTP 请求队列。长轮询请求可以被中断, 但是被代理的请求却不能。Netty 使服务于多种类型的请求很轻松。

- 静态的 HTML 页面——缓存的内容, 可以直接返回而不需要进行处理; 例子包括一个单页面的 HTTP 应用程序、robots.txt 和 crossdomain.xml。
- REST 请求——Firebase 支持传统的 GET、POST、PUT、DELETE、PATCH 以及 OPTIONS 请求。
- WebSocket——浏览器和 Firebase 服务器之间的双向连接, 拥有它自己的分帧协议。
- 长轮询——这些类似于 HTTP 的 GET 请求, 但是应用程序的处理方式有所不同。
- 被代理的请求——某些请求不能由接收它们的服务器处理。在这种情况下, Firebase 将会把这些请求代理到集群中正确的服务器。以便最终用户不必担心数据存储的具体位置。这些类似于 REST 请求, 但是代理服务器处理它们的方式有所不同。

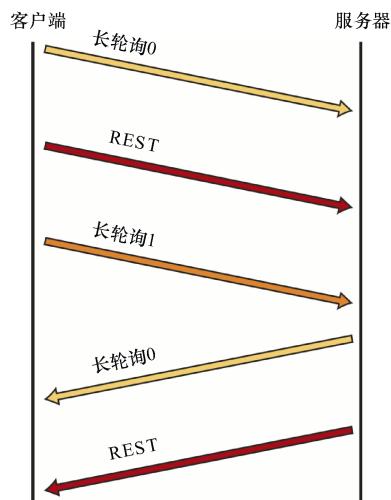


图 14-5 网络图

- 通过 SSL 的原始字节——一个简单的 TCP 套接字，运行 Firebase 自己的分帧协议，并且优化了握手过程。

Firebase 使用 Netty 来设置好它的 ChannelPipeline 以解析传入的请求，并随后适当地重新配置 ChannelPipeline 剩余的其他部分。在某些情况下，如 WebSocket 和原始字节，一旦某个特定类型的请求被分配给某个 Channel 之后，它就会在它的整个生命周期内保持一致。在其他情况下，如各种 HTTP 请求，该分配则必须以每个消息为基础进行赋值。同一个 Channel 可以处理 REST 请求、长轮询请求以及被代理的请求。

#### 14.2.4 控制 SslHandler

Netty 的 SslHandler 类是 Firebase 如何使用 Netty 来对它的网络通信进行细粒度控制的一个例子。当传统的 Web 技术栈使用 Apache 或者 Nginx 之类的 HTTP 服务器来将请求传递给应用程序时，传入的 SSL 请求在被应用程序的代码接收到的时候就已经被解码了。在多租户的架构体系中，很难将部分的加密流量分配给使用了某个特定服务的应用程序的租户。这很复杂，因为事实上多个应用程序可能使用了相同的加密 Channel 来和 Firebase 通信（例如，用户可能在不同的标签页中打开了两个 Firebase 应用程序）。为了解决这个问题，Firebase 需要在 SSL 请求被解码之前对它们拥有足够的控制来处理它们。

Firebase 基于带宽向客户进行收费。然而，对于某个消息来说，在 SSL 解密被执行之前，要收取费用的账户通常是不知道的，因为它被包含在加密了的有效负载中。Netty 使得 Firebase 可以在 ChannelPipeline 中的多个位置对流量进行拦截，因此对于字节数的统计可以从字节刚被从套接字读取出来时便立即开始。在消息被解密并且被 Firebase 的服务器端逻辑处理之后，字节计数便可以被分配给对应的账户。在构建这项功能时，Netty 在协议栈的每一层上，都提供了对于处理网络通信的控制，并且也使得非常精确的计费、限流以及速率限制成为了可能，所有的这一切都对业务具有显著的影响。

Netty 使得通过少量的 Scala 代码便可以拦截所有的人站消息和出站消息并且统计字节数成为了可能，如代码清单 14-3 所示。

代码清单 14-3 设置 ChannelPipeline

```
case class NamespaceTag(namespace: String)

class NamespaceBandwidthHandler extends ChannelDuplexHandler {
    private var rxBytes: Long = 0
    private var txBytes: Long = 0
    private var nsStats: Option[NamespaceStats] = None

    override def channelRead(ctx: ChannelHandlerContext, msg: Object) {
        msg match {
            case buf: ByteBuf =>
                rxBytes += buf.readableBytes(
                    tryFlush(ctx)
                )
        }
    }
}
```

当消息传入时，  
统计它的字节数

```

        }
        case _ => {
    }
    super.channelRead(ctx, msg)
}

override def write(ctx: ChannelHandlerContext, msg: Object,
    promise: ChannelPromise) {
    msg match {
        case buf: ByteBuf => {
            txBytes += buf.readableBytes()
            tryFlush(ctx)
            super.write(ctx, msg, promise)
        }
        case tag: NamespaceTag => {
            updateTag(tag.namespace, ctx)
        }
        case _ => {
            super.write(ctx, msg, promise)
        }
    }
}

private def tryFlush(ctx: ChannelHandlerContext) {
    nsStats match {
        case Some(stats: NamespaceStats) => {
            stats.logOutgoingBytes(txBytes.toInt) ←
            txBytes = 0
            stats.logIncomingBytes(rxBytes.toInt)
            rxBytes = 0
        }
        case None => {
            // no-op, we don't have a namespace
        }
    }
}

private def updateTag(ns: String, ctx: ChannelHandlerContext) {
    val (_, isLocalNamespace) = NamespaceOwnershipManager.getOwner(ns)
    if (isLocalNamespace) {
        nsStats = NamespaceStatsListManager.get(ns)
        tryFlush(ctx)
    } else {
        // Non-local namespace, just flush the bytes
        txBytes = 0
        rxBytes = 0
    }
}
}

当有出站消息时，同样统计这些字节数
如果接收到了命名空间标签，则将这个 Channel 关联到某个账户，记住该账户，并将当前的字节计数分配给它
如果已经有了该 Channel 所属的命名空间的标签，则将字节计数分配给该账户，并重置计数器
如果该字节计数不适用于这台机器，则忽略它并重置计数器

```

### 14.2.5 Firebase 小结

在 Firebase 的实时数据同步服务的服务器端架构中，Netty 扮演了不可或缺的角色。它使得可以支持一个异构的客户端生态系统，其中包括了各种各样的浏览器，以及完全由 Firebase 控制的客户端。使用 Netty，Firebase 可以在每个服务器上每秒钟处理数以万计的消息。Netty 之所以非常了不起，有以下几个原因。

- 它很快。开发原型只需要几天时间，并且从来不是生产瓶颈。
- 它的抽象层次具有良好的定位。Netty 提供了必要的细粒度控制，并且允许在控制流的每一步进行自定义。
- 它支持在同一个端口上支撑多种协议。HTTP、WebSocket、长轮询以及独立的 TCP 协议。
- 它的 GitHub 库是一流的。精心编写的 Javadoc 使得可以无障碍地利用它进行开发。
- 它拥有一个非常活跃的社区。社区非常积极地修复问题，并且认真地考虑所有的反馈以及合并请求。此外，Netty 团队还提供了优秀的最新的示例代码。Netty 是一个优秀的、维护良好的框架，而且它已经成为了构建和伸缩 Firebase 的基础设施的基础要素。如果没有 Netty 的速度、控制、抽象以及了不起的团队，那么 Firebase 中的实时数据同步将无从谈起。

## 14.3 Urban Airship——构建移动服务

### Erik Onnen，架构副总裁

随着智能手机的使用以前所未有的速度在全球范围内不断增长，涌现了大量的服务提供商，以协助开发者和市场人员提供令人惊叹不已的终端用户体验。不同于它们的功能手机前辈，智能手机渴求 IP 连接，并通过多个渠道（3G、4G、WiFi、WiMAX 以及蓝牙）来寻求连接。随着越来越多的这些设备通过基于 IP 的协议连接到公共网络，对于后端服务提供商来说，伸缩性、延迟以及吞吐量方面的挑战变得越来越艰巨了。

值得庆幸的是，Netty 非常适用于处理由随时在线的移动设备的惊群效应所带来的许多问题。本节将详细地介绍 Netty 在伸缩移动开发人员和市场人员平台——Urban Airship 时的几个实际应用。

### 14.3.1 移动消息的基础知识

虽然市场人员长期以来都使用 SMS 来作为一种触达移动设备的通道，但是最近一种被称为推送通知的功能正在迅速地成为向智能手机发送消息的首选机制。推送通知通常使用较为便宜的数据通道，每条消息的价格只是 SMS 费用的一小部分。推送通知的吞吐量通常都比 SMS 高 2~3 个数量级，所以它成为了突发新闻的理想通道。最重要的是，推送通知为用户提供了设备驱动的对推送通道的控制。如果一个用户不喜欢某个应用程序的通知消息，那么用户可以禁用该应用程序的通知，或者干脆删除该应用程序。

在一个非常高的级别上，设备和推送通知行为之间的交互类似于图 14-6 中所描述的那样。

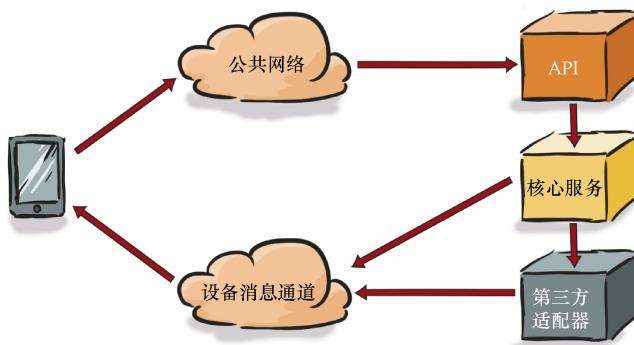


图 14-6 移动消息平台集成的高级别视图

在高级别上，当应用程序开发人员想要发送推送通知给某台设备时，开发人员必须要考虑存储有关设备及其应用程序安装的信息<sup>①</sup>。通常，应用程序的安装都将会执行代码以检索一个平台相关的标识符，并且将该标识符上报给一个持久化该标识符的中心化服务。稍后，应用程序安装之外的逻辑将会发起一个请求以向该设备投递一条消息。

一旦一个应用程序的安装已经将它的标识符注册到了后端服务，那么推送消息的递交就可以反过来采取两种方式。在第一种方式中，使用应用程序维护一条到后端服务的直接连接，消息可以被直接递交给应用程序本身。第二种方式更加常见，在这种方式中，应用程序将依赖第三方代表该后端服务来将消息递交给应用程序。在 Urban Airship，这两种递交推送通知的方式都有使用，而且也都大量地使用了 Netty。

### 14.3.2 第三方递交

在第三方推送递交的情况下，每个推送通知平台都为开发者提供了一个不同的 API，来将消息递交给应用程序安装。这些 API 有着不同的协议（基于二进制的或者基于文本的）、身份验证（OAuth、X.509 等）以及能力。对于集成它们并且达到最佳的吞吐量，每种方式都有着其各自不同的挑战。

尽管事实上每个这些提供商的根本目的都是向应用程序递交通知消息，但是它们各自又都采取了不同的方式，这对系统集成商造成了重大的影响。例如，苹果公司的 Apple 推送通知服务（APNS）定义了一个严格的二进制协议；而其他的提供商则将它们的服务构建在了某种形式的 HTTP 之上，所有的这些微妙变化都影响了如何以最佳的方式达到最大的吞吐量。值得庆幸的是，Netty 是一个灵活得令人惊奇的工具，它为消除不同协议之间的差异提供了极大的帮助。

<sup>①</sup> 某些移动操作系统允许一种被称为本地推送的推送通知，可能不会遵循这种做法。

接下来的几节将提供 Urban Airship 是如何使用 Netty 来集成两个上面所列出的服务提供商的例子。

### 14.3.3 使用二进制协议的例子

苹果公司的 APNS 是一个具有特定的网络字节序的有效载荷的二进制协议。发送一个 APNS 通知将涉及下面的事件序列：

- (1) 通过 SSLv3 连接将 TCP 套接字连接到 APNS 服务器，并用 X.509 证书进行身份认证；
- (2) 根据 Apple 定义的格式<sup>①</sup>，构造推送消息的二进制表示形式；
- (3) 将消息写出到套接字；
- (4) 如果你已经准备好了确定任何和已经发送的消息相关的错误代码，则从套接字中读取；
- (5) 如果有错误发生，则重新连接该套接字，并从步骤 2 继续。

作为格式化二进制消息的一部分，消息的生产者需要生成一个对于 APNS 系统透明的标识符。一旦消息无效（如不正确的格式、大小或者设备信息），那么该标识符将会在步骤 4 的错误响应消息中返回给客户端。

虽然从表面上看，该协议似乎简单明了，但是想要成功地解决所有上述问题，还是有一些微妙的细节，尤其是在 JVM 上。

- APNS 规范规定，特定的有效载荷值需要以大端字节序进行发送（如令牌长度）。
- 在前面的操作序列中的第 3 步要求两个解决方案二选一。因为 JVM 不允许从一个已经关闭的套接字中读取数据，即使在输出缓冲区中有数据存在，所以你有两个选项。
  - ◆ 在一次写出操作之后，在该套接字上执行带有超时的阻塞读取动作。这种方式有多个缺点，具体如下。
    - 阻塞等待错误消息的时间长短是不确定的。错误可能会发生在数毫秒或者数秒之内。
    - 由于套接字对象无法在多个线程之间共享，所以在等待错误消息时，对套接字的写操作必须立即阻塞。这将对吞吐量造成巨大的影响。如果在一次套接字写操作中递交单个消息，那么在直到读取超时发生之前，该套接字上都不会发出更多的消息。当你要递交数千万的消息时，每个消息之间都有 3 秒的延迟是无法接受的。
    - 依赖套接字超时是一项昂贵的操作。它将导致一个异常被抛出，以及几个不必要的系统调用。
  - ◆ 使用异步 I/O。在这个模型中，读操作和写操作都不会阻塞。这使得写入者可以持续地给 APNS 发送消息，同时也允许操作系统在数据可供读取时通知用户代码。

Netty 使得可以轻松地解决所有的这些问题，同时提供了令人惊叹的吞吐量。

首先，让我们看看 Netty 是如何简化使用正确的字节序打包二进制 APNS 消息的，如代码清

---

<sup>①</sup> 有关 APNS 的信息，参考 <http://docs.aws.amazon.com/sns/latest/dg/mobile-push-apns.html> 和 <http://bit.ly/189mmpG>。

单 14-4 所示。

#### 代码清单 14-4 ApnsMessage 实现

```

public final class ApnsMessage {
    private static final byte COMMAND = (byte) 1;
    public ByteBuf toBuffer() {
        short size = (short) (1 + // Command
            4 + // Identifier           因为消息的大小不一, 所以
            4 + // Expiry                出于效率考虑, 在 ByteBuf
            2 + // DT length header     创建之前将先计算它
            32 + // DS length
            2 + // body length header
            body.length);
        // 在创建时, ByteBuf 的大小正好, 并且指定了用于 APNS 的大端字节序
        ByteBuf buf = Unpooled.buffer(size).order(ByteOrder.BIG_ENDIAN);
        buf.writeByte(COMMAND);
        buf.writeInt(identifier);
        buf.writeInt(expiryTime);
        buf.writeShort((short) deviceToken.length);
        buf.writeBytes(deviceToken);
        buf.writeShort((short) body.length);
        buf.writeBytes(body);
        return buf;                  // 来自于类中其他地方维护的状态的各种值将会
    }                                // 被写入到缓冲区中
}
}                                // 这个类中的 deviceToken
                                  // 字段 (这里未展示) 是一个 Java 的 byte[]

```

当缓冲区已经就绪时, 简单地将它返回

关于该实现的一些重要说明如下。

- ① Java 数组的长度属性值始终是一个整数。但是, APNS 协议需要一个 2-byte 值。在这种情况下, 有效负载的长度已经在其他的地方验证过了, 所以在这里将其强制转换为 short 是安全的。注意, 如果没有显式地将 ByteBuf 构造为大端字节序, 那么在处理 short 和 int 类型的值时则可能会出现各种微妙的错误。
- ② 不同于标准的 `java.nio.ByteBuffer`, 没有必要翻转<sup>①</sup>缓冲区, 也没必要关心它的位置——Netty 的 `ByteBuf` 将会自动管理用于读取和写入的位置。

使用少量的代码, Netty 已经使得创建一个格式正确的 APNS 消息的过程变成小事一桩了。因为这个消息现在已经被打包进了一个 `ByteBuf`, 所以当消息准备好发送时, 便可以很容易地被直接写入连接了 APNS 的 `Channel`。

可以通过多重机制连接 APNS, 但是最基本的, 是需要一个使用 `SslHandler` 和解码器来填充 `ChannelPipeline` 的 `ChannelInitializer`, 如代码清单 14-5 所示。

<sup>①</sup> 即调用 `ByteBuffer` 的 `flip()` 方法。——译者注

## 代码清单 14-5 设置 ChannelPipeline

```

public final class ApnsClientPipelineInitializer
    extends ChannelInitializer<Channel> {
    private final SSLEngine clientEngine;

    public ApnsClientPipelineFactory(SSLEngine engine) {
        this.clientEngine = engine;
    }

    @Override
    public void initChannel(Channel channel) throws Exception {
        final ChannelPipeline pipeline = channel.pipeline();
        final SslHandler handler = new SslHandler(clientEngine);
        handler.setEnableRenegotiation(true);
        pipeline.addLast("ssl", handler);
        pipeline.addLast("decoder", new ApnsResponseDecoder());
    }
}

```

值得注意的是，Netty 使得协商结合了异步 I/O 的 X.509 认证的连接变得多么的容易。在 Urban Airship 早期的没有使用 Netty 的原型 APNS 的代码中，协商一个异步的 X.509 认证的连接需要 80 多行代码和一个线程池，而这只仅仅是为了建立连接。Netty 隐藏了所有的复杂性，包括 SSL 握手、身份验证、最重要的将明文的字节加密为密文，以及使用 SSL 所带来的密钥的重新协商。这些 JDK 中异常无聊的、容易出错的并且缺乏文档的 API 都被隐藏在了 3 行 Netty 代码之后。

在 Urban Airship，在所有和众多的包括 APNS 以及 Google 的 GCM 的第三方推送通知服务的连接中，Netty 都扮演了重要的角色。在每种情况下，Netty 都足够灵活，允许显式地控制从更高级别的 HTTP 的连接行为到基本的套接字级别的配置（如 TCP keep-alive 以及套接字缓冲区大小）的集成如何生效。

#### 14.3.4 直接面向设备的递交

上一节提供了 Urban Airship 如何与第三方集成以进行消息递交的内部细节。在谈及图 14-6 时，需要注意的是，将消息递交到设备有两种方式。除了通过第三方来递交消息之外，Urban Airship 还有直接作为消息递交通道的经验。在作为这种角色时，单个设备将直接连接 Urban Airship 的基础设施，绕过第三方提供商。这种方式也带来了一组截然不同的挑战。

- 由移动设备发出的套接字连接往往是短暂的。根据不同的条件，移动设备将频繁地在不同类型的网络之间进行切换。对于移动服务的后端提供商来说，设备将不断地重新连接，并将感受到短暂而又频繁的连接周期。

- 跨平台的连接性是不规则的。从网络的角度来看，平板设备的连接性往往表现得和移动电话不一样，而对比于台式计算机，移动电话的连接性的表现又不一样。
- 移动电话向后端服务提供商更新的频率一定会增加。移动电话越来越多地被应用于日常任务中，不仅产生了大量常规的网络流量，而且也为后端服务提供商提供了大量的分析数据。
- 电池和带宽不能被忽略。不同于传统的桌面环境，移动电话通常使用有限的数据流量包。服务提供商必须要尊重最终用户只有有限的电池使用时间，而且他们使用昂贵的、速率有限的（蜂窝移动数据网络）带宽这一事实。滥用两者之一都通常会导致应用被卸载，这对于移动开发人员来说可能是最坏的结果了。
- 基础设施的所有方面都需要大规模的伸缩。随着移动设备普及程度的不断增加，更多的应用程序安装量将会导致更多的到移动服务的基础设施的连接。由于移动设备的庞大規模和增长，这个列表中的每一个前面提到的元素都将变得愈加复杂。

随着时间的推移，Urban Airship 从移动设备的不断增长中学到了几点关键的经验教训：

- 移动运营商的多样性可以对移动设备的连接性造成巨大的影响；
- 许多运营商都不允许 TCP 的 keep-alive 特性，因此许多运营商都会积极地剔除空闲的 TCP 会话；
- UDP 不是一个可行的向移动设备发送消息的通道，因为许多的运营商都禁止它；
- SSLv3 所带来的开销对于短暂的连接来说是巨大的痛苦。

鉴于移动增长的挑战，以及 Urban Airship 的经验教训，Netty 对于实现一个移动消息平台来说简直就是天作之合，原因将在以下各节强调。

### 14.3.5 Netty 擅长管理大量的并发连接

如上一节中所提到的，Netty使得可以轻松地在JVM平台上支持异步I/O。因为Netty运行在JVM之上，并且因为JVM在Linux上将最终使用Linux的epoll方面的设施来管理套接字文件描述符中所感兴趣的事件（interest），所以Netty使得开发者能够轻松地接受大量打开的套接字——每一个Linux进程将近一百万的TCP连接，从而适应快速增长的移动设备的规模。有了这样的伸缩能力，服务提供商便可以在保持低成本的同时，允许大量的设备连接到物理服务器上的一个单独的进程<sup>①</sup>。

在受控的测试以及优化了配置选项以使用少量的内存的条件下，一个基于 Netty 的服务得以容纳略少于 100 万（约为 998 000）的连接。在这种情况下，这个限制从根本上来说是由

---

① 注意，在这种情况下物理服务器的区别。尽管虚拟化提供了许多的好处，但是领先的云计算提供商仍然未能支持到单个虚拟主机超过 200 000 ~ 300 000 的并发 TCP 连接。当连接达到或者超过这种规模时，建议使用裸机（bare metal）服务器，并且密切关注网络接口卡（Network Interface Card，NIC）提供商。

于 Linux 内核强制硬编码了每个进程限制 100 万个文件句柄。如果 JVM 本身没有持有大量的套接字以及用于 JAR 文件的文件描述符，那么该服务器可能本能够处理更多的连接，而所有的这一切都在一个 4GB 大小的堆上。利用这种效能，Urban Airship 成功地维持了超过 2000 万的到它的基础设施的持久化的 TCP 套接字连接以进行消息递交，所有的这一切都只使用了少量的服务器。

值得注意的是，虽然在实践中，一个单一的基于 Netty 的服务便能够处理将近 1 百万的入站 TCP 套接字连接，但是这样做并不一定就是务实的或者明智的。如同分布式计算中的所有陷阱一样，主机将会失败、进程将需要重新启动并且将会发生不可预期的行为。由于这些现实的问题，适当的容量规划意味着需要考虑到单个进程失败的后果。

### 14.3.6 Urban Airship 小结——跨越防火墙边界

我们已经演示了两个在 Urban Airship 内部网络中每天都会使用 Netty 的场景。Netty 适合这些用途，并且工作得非常出色，但在 Urban Airship 内部的许多其他的组件中也有它作为脚手架存在的身影。

#### 1. 内部的 RPC 框架

Netty 一直都是 Urban Airship 内部的 RPC 框架的核心，其一直都在不断进化。今天，这个框架每秒钟可以处理数以十万计的请求，并且拥有相当低的延迟以及杰出的吞吐量。几乎每个由 Urban Airship 发出的 API 请求都经由了多个后端服务处理，而 Netty 正是所有这些服务的核心。

#### 2. 负载和性能测试

Netty 在 Urban Airship 已经被用于几个不同的负载测试框架和性能测试框架。例如，在测试前面所描述的设备消息服务时，为了模拟数百万的设备连接，Netty 和一个 Redis 实例 (<http://redis.io/>) 相结合使用，以最小的客户端足迹（负载）测试了端到端的消息吞吐量。

#### 3. 同步协议的异步客户端

对于一些内部的使用场景，Urban Airship 一直都在尝试使用 Netty 来为典型的同步协议创建异步的客户端，包括如 Apache Kafka(<http://kafka.apache.org/>) 以及 Memcached(<http://memcached.org/>) 这样的服务。Netty 的灵活性使得我们能够很容易地打造天然异步的客户端，并且能够在真正的异步或同步的实现之间来回地切换，而不需要更改任何的上游代码。

总而言之，Netty 一直都是 Urban Airship 服务的基石。其作者和社区都是极其出色的，并为任何需要在 JVM 上进行网络通信的应用程序，创造了一个真正意义上的一流框架。

## 14.4 小结

本章旨在揭示真实世界中的 Netty 的使用场景，以及它是如何帮助这些公司解决了重大的网络通信问题的。值得注意的是，在所有的场景下，Netty 都不仅是被作为一个代码框架而使用，而且还是开发和架构最佳实践的重要组成部分。

在下一章中，我们将介绍由 Facebook 和 Twitter 所贡献的案例研究，描述两个开源项目，这两个项目是从基于 Netty 的最初被开发用来满足内部需求的项目演化而来的。

# 第 15 章 案例研究，第二部分

## 本章主要内容

- Facebook 的案例研究
- Twitter 的案例研究

在本章中，我们将看到 Facebook 和 Twitter（两个最流行的社交网络）是如何使用 Netty 的。他们都利用了 Netty 灵活和通用的设计来构建框架和服务，以满足对极端伸缩性以及可扩展性的需求。

这里所呈现的案例研究都是由那些负责设计和实现所述解决方案的工程师所撰写的。

## 15.1 Netty在Facebook的使用：Nifty和Swift<sup>①</sup>

Andrew Cox, Facebook 软件工程师

在 Facebook，我们在我们的几个后端服务中使用了 Netty（用于处理来自手机应用程序的消息通信、用于 HTTP 客户端等），但是我们增长最快的用法还是通过我们所开发的用来构建 Java 的 Thrift 服务的两个新框架：Nifty 和 Swift。

### 15.1.1 什么是 Thrift

Thrift 是一个用来构建服务和客户端的框架，其通过远程过程调用（RPC）来进行通信。它最初是在 Facebook 开发的<sup>②</sup>，用以满足我们构建能够处理客户端和服务器之间的特定类型的接口不匹配的服务的需要。这种方式十分便捷，因为服务器和它们的客户端通常不能全部同时升级。

① 本节所表达的观点都是本节作者的观点，并不一定反映了该作者的雇主的观点。

② 一份来自原始的 Thrift 的开发者的不旧不新的白皮书可以在 <http://thrift.apache.org/static/files/thrift-20070401.pdf> 找到。

Thrift 的另一个重要的特点是它可以被用于多种语言。这使得在 Facebook 的团队可以为工作选择正确的语言，而不必担心他们是否能够找到和其他的服务相互交互的客户端代码。在 Facebook，Thrift 已经成为我们的后端服务之间相互通信的主要方式之一，同时它还被用于非 RPC 的序列化任务，因为它提供了一个通用的、紧凑的存储格式，能够被多种语言读取，以便后续处理。

自从Thrift在Facebook被开发以来，它已经作为一个Apache项目（<http://thrift.apache.org/>）开源了，在那里它将继续成长以满足服务开发人员的需要，不止在Facebook有使用，在其他公司也有使用，如Evernote和last.fm<sup>①</sup>，以及主要的开源项目如Apache Cassandra和HBase等。

下面是 Thrift 的主要组件：

- Thrift 的接口定义语言 (IDL) ——用来定义你的服务，并且编排你的服务将要发送和接收的任何自定义类型；
- 协议——用来控制将数据元素编码/解码为一个通用的二进制格式（如 Thrift 的二进制协议或者 JSON）；
- 传输——提供了一个用于读/写不同媒体（如 TCP 套接字、管道、内存缓冲区）的通用接口；
- Thrift 编译器——解析 Thrift 的 IDL 文件以生成用于服务器和客户端的存根代码，以及在 IDL 中定义的自定义类型的序列化/反序列化代码；
- 服务器实现——处理接受连接、从这些连接中读取请求、派发调用到实现了这些接口的对象，以及将响应发回给客户端；
- 客户端实现——将方法调用转换为请求，并将它们发送给服务器。

### 15.1.2 使用 Netty 改善 Java Thrift 的现状

Thrift 的 Apache 分发版本已经被移植到了大约 20 种不同的语言，而且还有用于其他语言的和 Thrift 相互兼容的独立框架（Twitter 的用于 Scala 的 Finagle 便是一个很好的例子）。这些语言中的一些在 Facebook 多多少少有被使用，但是在 Facebook 最常用的用来编写 Thrift 服务的还是 C++ 和 Java。

当我加入Facebook时，我们已经在使用C++围绕着libevent，顺利地开发可靠的、高性能的、异步的Thrift实现了。通过libevent，我们得到了OS API之上的跨平台的异步I/O抽象，但是libevent并不会比，比如说，原始的Java NIO，更加容易使用。因此，我们也在其上构建了抽象，如异步的消息通道，同时我们还使用了来自Folly<sup>②</sup>的链式缓冲区尽可能地避免复制。这个框架还具有一个支持带有多路复用的异步调用的客户端实现，以及一个支持异步的请求处理的服务器实现。（该

① 可以在 <http://thrift.apache.org> 找到更多的例子。

② Folly 是 Facebook 的开源 C++ 公共库：<https://www.facebook.com/notes/facebook-engineering/folly-the-facebook-open-source-library/10150864656793920>。

服务器可以启动一个异步任务来处理请求并立即返回，随后在响应就绪时调用一个回调或者稍后设置一个Future。)

同时，我们的 Java Thrift 框架却很少受到关注，而且我们的负载测试工具显示 Java 版本的性能明显落后于 C++ 版本。虽然已经有了构建于 NIO 之上的 Java Thrift 框架，并且异步的基于 NIO 的客户端也可用。但是该客户端不支持流水线化以及请求的多路复用，而服务器也不支持异步的请求处理。由于这些缺失的特性，在 Facebook，这里的 Java Thrift 服务开发人员遇到了那些在 C++（的 Thrift 框架）中已经解决了的问题，并且它也成为了挫败感的源泉。

我们本来可以在 NIO 之上构建一个类似的自定义框架，并在那之上构建我们新的 Java Thrift 实现，就如同我们为 C++ 版本的实现所做的一样。但是经验告诉我们，这需要巨大的工作量才能完成，不过碰巧，我们所需要的框架已经存在了，只等着我们去使用它：Netty。

我们很快地组装了一个服务器实现，并且将名字“Netty”和“Thrift”混在一起，为新的服务器实现提出了“Nifty”这个名字。相对于在 C++ 版本中达到同样的效果我们所需要做的一切，那么少的代码便可以使得 Nifty 工作，这立即就让人印象深刻。

接下来，我们使用 Nifty 构建了一个简单的用于负载测试的 Thrift 服务器，并且使用我们的负载测试工具，将它和我们现有的服务器进行了对比。结果是显而易见的：Nifty 的表现要优于其他的 NIO 服务器，而且和我们最新的 C++ 版本的 Thrift 服务器的结果也不差上下。使用 Netty 就是为了要提高性能！

### 15.1.3 Nifty 服务器的设计

Nifty (<https://github.com/facebook/nifty>) 是一个开源的、使用 Apache 许可的、构建于 Apache Thrift 库之上的 Thrift 客户端/服务器实现。它被专门设计，以便无缝地从任何其他的 Java Thrift 服务器实现迁移过来：你可以重用相同的 Thrift IDL 文件、相同的 Thrift 代码生成器（与 Apache Thrift 库打包在一起），以及相同的服务接口实现。唯一真正需要改变的只是你的服务器的启动代码（Nifty 的设置风格与 Apache Thrift 中的传统的 Thrift 服务器实现稍微有所不同）。

#### 1. Nifty 的编码器/解码器

默认的 Nifty 服务器能处理普通消息或者分帧消息（带有 4 字节的前缀）。它通过使用自定义的 Netty 帧解码器做到了这一点，其首先查看前几个字节，以确定如何对剩余的部分进行解码。然后，当发现了一个完整的消息时，解码器将会把消息的内容和一个指示了消息类型的字段包装在一起。服务器随后将会根据该字段来以相同的格式对响应进行编码。

Nifty 还支持接驳你自己的自定义编解码器。例如，我们的一些服务使用了自定义的编解码器来从客户端在每条消息前面所插入的头部中读取额外的信息（包含可选的元数据、客户端的能力等）。解码器也可以被方便地扩展以处理其他类型的消息传输，如 HTTP。

## 2. 在服务器上排序响应

Java Thrift 的初始版本使用了 OIO 套接字，并且服务器为每个活动连接都维护了一个线程。使用这种设置，在下一个响应被读取之前，每个请求都将在同一个线程中被读取、处理和应答。这保证了响应将总会以对应的请求所到达的顺序返回。

较新的异步 I/O 的服务器实现诞生了，其不需要每个连接一个线程，而且这些服务器可以处理更多的并发连接，但是客户端仍然主要使用同步 I/O，因此服务器可以期望它在发送当前响应之前，不会收到下一个请求。这个请求/执行流如图 15-1 所示。

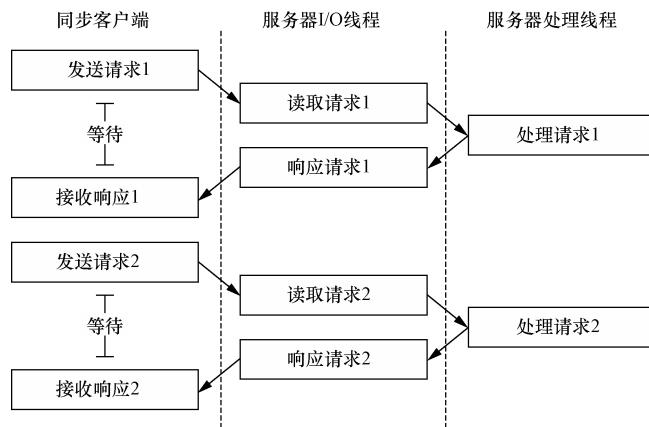


图 15-1 同步的请求/响应流

客户端最初的伪异步用法开始于一些 Thrift 用户利用的一项事实：对于一个生成的客户端方法 `foo()` 来说，方法 `send_foo()` 和 `recv_foo()` 将会被单独暴露出来。这使得 Thrift 用户可以发送多个请求（无论是在多个客户端上，还是在同一个客户端上），然后调用相应的接收方法来开始等待并收集结果。

在这个新的场景下，服务器可能会在它完成处理第一个请求之前，从单个客户端读取多个请求。在一个理想的世界中，我们可以假设所有流水线化请求的异步 Thrift 客户端都能够处理以任意顺序到达的这些请求所对应的响应。然而，在现实生活中，虽然新的客户端可以处理这种情况，而那些旧一点的异步 Thrift 客户端可能会写出多个请求，但是必须要求按顺序接收响应。

这种问题可以通过使用 Netty 4 的 `EventExecutor` 或者 Netty 3.x 中的 `OrderedMemoryAwareThreadPoolExecutor` 解决，其能够保证顺序地处理同一个连接上的所有传入消息，而不会强制所有这些消息都在同一个执行器线程上运行。

图 15-2 展示了流水线化的请求是如何被以正确的顺序处理的，这也就意味着对应于第一个请求的响应将会被首先返回，然后是对应于第二个请求的响应，以此类推。

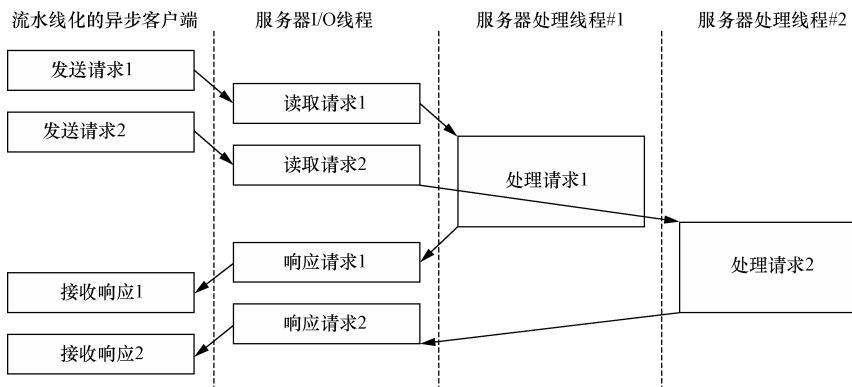


图 15-2 对于流水线化的请求的顺序化处理的请求/响应流

尽管 Nifty 有着特殊的要求：我们的目标是以客户端能够处理的最佳的响应顺序服务于每个客户端。我们希望允许用于来自于单个连接上的多个流水线化的请求的处理器能够被并行处理，但是那样我们又控制不了这些处理器完成的先后顺序。

相反，我们使用了一种涉及缓冲响应的方案；如果客户端要求保持顺序的响应，我们将会缓冲后续的响应，直到所有较早的响应也可用，然后我们将按照所要求的顺序将它们一起发送出去。见图 15-3 所示。

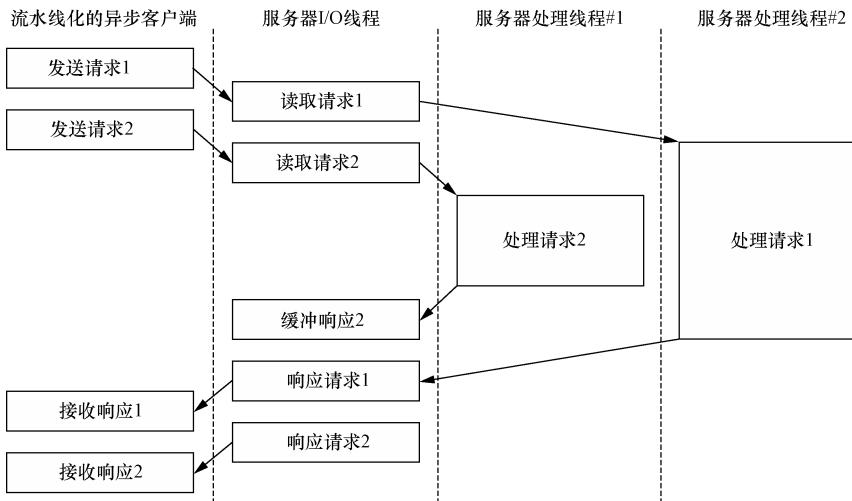


图 15-3 对于流水线化的请求的并行处理的请求/响应流

当然，Nifty 包括了实实在在支持无序响应的异步 Channel（可以通过 Swift 使用）。当使用能够让客户端通知服务器此客户端的能力的自定义的传输时，服务器将会免除缓冲响应的负担，并且将以请求完成的任意顺序把它们发送回去。

### 15.1.4 Nifty 异步客户端的设计

Nifty 的客户端开发主要集中在异步客户端上。Nifty 实际上也提供了一个针对 Thrift 的同步传输接口的 Netty 实现，但是它的使用相当受限，因为相对于 Thrift 所提供的标准的套接字传输，它并没有太多的优势。因此，用户应该尽可能地使用异步客户端。

#### 1. 流水线化

Thrift 库拥有它自己的基于 NIO 的异步客户端实现，但是我们想要的一个特性是请求的流水线化。流水线化是一种在同一连接上发送多个请求，而不需要等待其响应的能力。如果服务器有空闲的工作线程，那么它便可以并行地处理这些请求，但是即使所有的工作线程都处于忙绿状态，流水线化仍然可以有其他方面的裨益。服务器将会花费更少的时间来等待读取数据，而客户端则可以在一个单一的 TCP 数据包里一起发送多个小请求，从而更好地利用网络带宽。

使用 Netty，流水线化水到渠成。Netty 做了所有管理各种 NIO 选择键的状态的艰涩的工作，Nifty 则可以专注于解码请求以及编码响应。

#### 2. 多路复用

随着我们的基础设施的增长，我们开始看到在我们的服务器上建立起来了大量的连接。多路复用（为所有的连接来自于单一的来源的 Thrift 客户端共享连接）可以帮助减轻这种状况。但是在需要按序响应的客户端连接上进行多路复用会导致一个问题：该连接上的客户端可能会招致额外的延迟，因为它的响应必须要跟在对应于其他共享该连接的请求的响应之后。

基本的解决方案也相当简单：Thrift 已经在发送每个消息时都捎带了一个序列标识符，所以为了支持无序响应，我们只需要客户端 Channel 维护一个从序列 ID 到响应处理器的一个映射，而不是一个使用队列。

但是问题的关键在于，在标准的同步 Thrift 客户端中，协议层将负责从消息中提取序列标识符，再由协议层协议调用传输层，而不是其他的方式。

对于同步客户端来说，这种简单的流程（如图 15-4 所示）能够良好地工作，其协议层可以在传输层上等待，以实际接收响应，但是对于异步客户端来说，其控制流就变得更加复杂了。客户端调用将会被分发到 Swift 库中，其将首先要求协议层将请求编码到一个缓冲区，然后将编码请求缓冲区传递给 Nifty 的 Channel 以便被写出。当该 Channel 收到来自服务器的响应时，它将会通知 Swift 库，其将再次使用协议层以对响应缓冲区进行解码。这就是图 15-5 中所展示的流程。

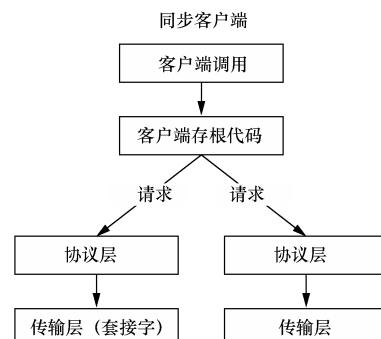


图 15-4 多路复用/传输层

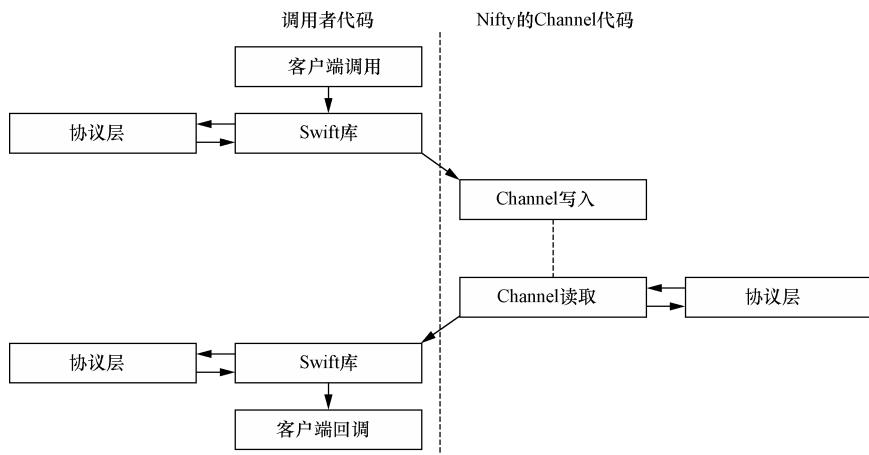


图 15-5 派发

### 15.1.5 Swift：一种更快的构建 Java Thrift 服务的方式

我们新的 Java Thrift 框架的另一个关键部分叫作 Swift。它使用了 Nifty 作为它的 I/O 引擎，但是其服务规范可以直接通过 Java 注解来表示，使得 Thrift 服务开发人员可以纯粹地使用 Java 进行开发。当你的服务启动时，Swift 运行时将通过组合使用反射以及解析 Swift 的注解来收集所有相关服务以及类型的信息。通过这些信息，它可以构建出和 Thrift 编译器在解析 Thrift IDL 文件时构建的模型一样的模型。然后，它将使用这个模型，并通过从字节码生成用于序列化/反序列化这些自定义类型的新类，来直接运行服务器以及客户端（而不需要任何生成的服务器或者客户端的存根代码）。

跳过常规的 Thrift 代码生成，还能使添加新功能变得更加轻松，而无需修改 IDL 编译器，所以我们的许多新功能（如异步客户端）都是首先在 Swift 中得到支持。如果你感兴趣，可以查阅 Swift 的 GitHub 页面 (<https://github.com/facebook/swift>) 上的介绍信息。

### 15.1.6 结果

在下面的各节中，我们将量化一些我们使用 Netty 的过程中所观察到的一些成果。

#### 1. 性能比较

一种测量 Thrift 服务器性能的方式是对空操作的基准测试。这种基准测试使用了长时间运行的客户端，这些客户端不间断地对发送回空响应的服务器进行 Thrift 调用。虽然这种测量方式对于大部分的实际 Thrift 服务来说，不是真实意义上的性能测试，但是它仍然很好地度量了 Thrift 服务的最大潜能，而且提高这一基准，通常也就意味着减少了该框架本身的 CPU 使用。

如表 15-1 所示，在这个基准测试下，Nifty 的性能优于所有其他基于 NIO 的 Thrift 服务器（TNonblockingServer、TThreadedSelectorServer 以及 TThreadPoolServer）的实现。它甚至轻松地击败了我们以前的 Java 服务器实现（我们内部使用的一个 Nifty 之前的服务器实现，基于原始的 NIO 以及直接缓冲区）。

表 15-1 不同实现的基准测试结果

Thrift 服务器实现	空操作请求/秒
TNonblockingServer	~68 000
TThreadedSelectorServer	188 000
TThreadPoolServer	867 000
较老的 Java 服务器（使用 NIO 和直接缓冲区）	367 000
Nifty	963 000
较老的基于 libevent 的 C++ 服务器	895 000
下一代基于 libevent 的 C++ 服务器	1 150 000

我们所测试过的唯一能够和 Nifty 相提并论的 Java 服务器是 TThreadPoolServer。这个服务器实现使用了原始的 OIO，并且在一个专门的线程上运行每个连接。这使得它在处理少量的连接时表现不错；然而，使用 OIO，当你的服务器需要处理大量的并发连接时，你将很容易遇到伸缩性问题。

Nifty 甚至击败了之前的 C++ 服务器实现，这是我们开始开发 Nifty 时最夺目的一点，虽然它相对于我们的下一代 C++ 服务器框架还有一些差距，但至少也大致相当。

## 2. 稳定性问题的例子

在 Nifty 之前，我们在 Facebook 的许多主要的 Java 服务都使用了一个较老的、自定义的基于 NIO 的 Thrift 服务器实现，它的工作方式类似于 Nifty。该实现是一个较旧的代码库，有更多的时间成熟，但是由于它的异步 I/O 处理代码是从零开始构建的，而且因为 Nifty 是构建在 Netty 的异步 I/O 框架的坚实基础之上的，所以（相比之下）它的问题也就少了很多。

我们的一个自定义的消息队列服务是基于那个较旧的框架构建的，而它开始遭受一种套接字泄露。大量的连接都停留在了 CLOSE\_WAIT 状态，这意味着服务器接收了客户端已经关闭了套接字的通知，但是服务器从来不通过其自身的调用来关闭套接字进行回应。这使得这些套接字都停滞在了 CLOSE\_WAIT 状态。

问题发生得很慢；在处理这个服务的整个机器集群中，每秒可能有数以百万计的请求，但是通常在一个服务器上只有一个套接字会在一个小时之内进入这个状态。这不是一个迫在眉睫的问题，因为在那种速率下，在一个服务器需要重启前，将需要花费很长的时间，但是这也复杂化了追查原因的过程。彻底地挖掘代码也没有带来太大的帮助：最初的几个地方看起来可疑，但是最终都被排除了，而我们也并没有定位到问题所在。

最终，我们将该服务迁移到了 Nifty 之上。转换（包括在预发环境中进行测试）花了不到一天的时间，而这个问题就此消失了。使用 Nifty，我们就真的再也没见过类似的问题了。

这只是在直接使用 NIO 时可能会出现的微妙 bug 的一个例子，而且它类似于那些在我们的 C++ Thrift 框架稳定的过程中，不得不一次又一次地解决的 bug。但是我认为这是一个很好的例子，它说明了通过使用 Netty 是如何帮助我们利用它多年来收到的稳定性修复的。

### 3. 改进 C++实现的超时处理

Netty 还通过为改进我们的 C++ 框架提供一些启发间接地帮助了我们。一个这样的例子是基于散列轮的计时器。我们的 C++ 框架使用了来自于 libevent 的超时事件来驱动客户端以及服务器的超时，但是为每个请求都添加一个单独的超时被证明是十分昂贵的，因此我们一直都在使用我们称之为超时集的东西。其思想是：一个到特定服务的客户端连接，对于由该客户端发出的每个请求，通常都具有相同的接收超时，因此对于一组共享了相同的时间间隔的超时集合，我们仅维护一个真正的计时器事件。每个新的超时都将被保证会在对于该超时集合的现存的超时被调度之后触发，因此当每个超时过期或者被取消时，我们将只会安排下一个超时。

然而，我们的用户偶尔想要为每个调用都提供单独的超时，在相同连接上的不同的请求设置不同的超时值。在这种情况下，使用超时集合的好处就消失了，因此我们尝试了使用单独的计时器事件。在大量的超时被同时调度时，我们开始看到了性能问题。我们知道 Nifty 不会碰到这个问题，除了它不使用超时集的这个事实——Netty 通过它的 HashedWheelTimer<sup>①</sup>解决了该问题。因此，带着来自 Netty 的灵感，我们为我们的 C++ Thrift 框架添加了一个基于散列轮的计时器，并解决了可变的每请求（per-request）超时时间间隔所带来的性能问题。

### 4. 未来基于 Netty 4 的改进

Nifty 目前运行在 Netty 3 上，这对我们来说已经很好了，但是我们已经有一个基于 Netty 4 的移植版本准备好了，现在第 4 版的 Netty 已经稳定下来了，我们很快就会迁移过去。我们热切地期待着 Netty 4 的 API 将会带给我们的一些益处。

一个我们计划如何更好地利用 Netty 4 的例子是实现更好地控制哪个线程将管理一个给定的连接。我们希望使用这项特性，可以使服务器的处理器方法能够从和该服务器调用所运行的 I/O 线程相同的线程开始异步的客户端调用。这是那些专门的 C++ 服务器（如 Thrift 请求路由器）已经能够利用的特性。

从该例子延伸开来，我们也期待着能够构建更好的客户端连接池，使得能够把现有的池化连接迁移到期望的 I/O 工作线程上，这在第 3 版的 Netty 中是不可能做到的。

---

① 有关 HashedWheelTimer 类的更多的信息，参见 <http://netty.io/4.1/api/io/netty/util/HashedWheelTimer.html>。

### 15.1.7 Facebook 小结

在 Netty 的帮助下，我们已经能够构建更好的 Java 服务器框架了，其几乎能够与我们最快的 C++ Thrift 服务器框架的性能相媲美。我们已经将我们现有的一些主要的 Java 服务迁移到了 Nifty，并解决了一些令人讨厌的稳定性和性能问题，同时我们还开始将一些来自 Netty，以及 Nifty 和 Swift 开发过程中的思想，反馈到提高 C++ Thrift 的各个方面中。

不仅如此，使用 Netty 是令人愉悦的，并且它已经添加了大量的新特性，例如，对于 Thrift 客户端的内置 SOCKS 支持来说，添加起来小菜一碟。

但是我们并不止步于此。我们还有大量的性能调优工作要做，以及针对将来的大量的其他方面的改进计划。如果你对使用 Java 进行 Thrift 开发感兴趣，一定要关注哦！

## 15.2 Netty 在 Twitter 的使用：Finagle

Jeff Smick，Twitter 软件工程师

Finagle 是 Twitter 构建在 Netty 之上的容错的、协议不可知的 RPC 框架。从提供用户信息、推特以及时间线的后端服务到处理 HTTP 请求的前端 API 端点，所有组成 Twitter 架构的核心服务都建立在 Finagle 之上。

### 15.2.1 Twitter 成长的烦恼

Twitter 最初是作为一个整体式的 Ruby On Rails 应用程序构建的，我们半亲切地称之为 Monorail。随着 Twitter 开始经历大规模的成长，Ruby 运行时以及 Rails 框架开始成为瓶颈。从计算机的角度来看，Ruby 对资源的利用是相对低效的。从开发的角度来看，该 Monorail 开始变得难以维护。对一个部分的代码修改将会不透明地影响到另外的部分。代码的不同方面的所属权也不清楚。无关核心业务对象的小改动也需要一次完整的部署。核心业务对象也没有暴露清晰的 API，其加剧了内部结构的脆弱性以及发生故障的可能性。

我们决定将该 Monorail 分拆为不同的服务，明确归属人并且精简 API，使迭代更快速，维护更容易。每个核心业务对象都将由一个专门的团队维护，并且由它自己的服务提供支撑。公司内部已经有了在 JVM 上进行开发的先例——几个核心的服务已经从该 Monorail 中迁移出去，并已经用 Scala 重写了。我们的运维团队也有运维 JVM 服务的背景，并且知道如何运维它们。鉴于此，我们决定使用 Java 或者 Scala 在 JVM 上构建所有的新服务。大多数的服务开发团队都决定选用 Scala 作为他们的 JVM 语言。

### 15.2.2 Finagle 的诞生

为了构建出这个新的架构，我们需要一个高性能的、容错的、协议不可知的、异步的 RPC 框架。

在面向服务的架构中，服务花费了它们大部分的时间来等待来自其他上游的服务的响应。使用异步的库使得服务可以并发地处理请求，并且充分地利用硬件资源。尽管 Finagle 可以直接建立在 NIO 之上，但是 Netty 已经解决了许多我们可能会遇到的问题，并且它提供了一个简洁、清晰的 API。

Twitter 构建在几种开源的协议之上，主要是 HTTP、Thrift、Memcached、MySQL 以及 Redis。我们的网络栈需要具备足够的灵活性，能够和任何的这些协议进行交流，并且具备足够的可扩展性，以便我们可以方便地添加更多的协议。Netty 并没有绑定任何特定的协议。向它添加协议就像创建适当的 ChannelHandler 一样简单。这种扩展性也催生了许多社区驱动的协议实现，包括 SPDY、PostgreSQL、WebSockets、IRC 以及 AWS<sup>①</sup>。

Netty 的连接管理以及协议不可知的特性为构建 Finagle 提供了绝佳的基础。但是我们也有一些其他的 Netty 不能开箱即满足的需求，因为那些需求都更高级。客户端需要连接到服务器集群，并且需要做跨服务器集群的负载均衡。所有的服务都需要暴露运行指标（请求率、延迟等），其可以为调试服务的行为提供有价值的数据。在面向服务的架构中，一个单一的请求都可能需要经过数十种服务，使得如果没有一个由 Dapper 启发的跟踪框架，调试性能问题几乎是不可能的<sup>②</sup>。Finagle 正是为了解决这些问题而构建的。

### 15.2.3 Finagle 是如何工作的

Finagle 的内部结构是非常模块化的。组件都是先独立编写，然后再堆叠在一起。根据所提供的配置，每个组件都可以被换入或者换出。例如，所有的跟踪器都实现了相同的接口，因此可以创建一个跟踪器用来将跟踪数据发送到本地文件、保存在内存中并暴露一个读取端点，或者将它写出到网络。

在 Finagle 栈的底部是 Transport 层。这个类表示了一个能够被异步读取和写入的对象流。Transport 被实现为 Netty 的 ChannelHandler，并被插入到了 ChannelPipeline 的尾端。来自网络的消息在被 Netty 接收之后，将经由 ChannelPipeline，在那里它们将被编解码器解释，并随后被发送到 Finagle 的 Transport 层。从那里 Finagle 将会从 Transport 层读取消息，并且通过它自己的栈发送消息。

对于客户端的连接，Finagle 维护了一个可以在其中进行负载均衡的传输（Transport）池。根据所提供的连接池的语义，Finagle 将从 Netty 请求一个新连接或者复用一个现有的连接。当请求新连接时，将会根据客户端的编解码器创建一个 Netty 的 ChannelPipeline。额外的用于统计、日志记录以及 SSL 的 ChannelHandler 将会被添加到该 ChannelPipeline 中。该连接随后将会

<sup>①</sup> 关于 SPDY 的更多信息参见 <https://github.com/twitter/finagle/tree/master/finagle-spdy>。关于 PostgreSQL 参见 <https://github.com/mairbek/finagle-postgres>。关于 WebSockets 参见 <https://github.com/sprsquish/finagle-websocket>。关于 IRC 参见 <https://github.com/sprsquish/finagle-irc>。关于 AWS 参见 <https://github.com/sclasen/finagle-aws>。

<sup>②</sup> 有关 Dapper 的信息可以在 <http://research.google.com/pubs/pub36356.html> 找到。该分布式的跟踪框架是 Zipkin，可以在 <https://github.com/twitter/zipkin> 找到。

被递给一个Finagle能够写入和读取的ChannelTransport<sup>①</sup>。

在服务器端，创建了一个Netty服务器，然后向其提供一个管理编解码器、统计、超时以及日志记录的ChannelPipelineFactory。位于服务器ChannelPipeline尾端的ChannelHandler是一个Finagle的桥接器。该桥接器将监控新的传入连接，并为每一个传入连接创建一个新的Transport。该Transport将在新的Channel被递交给某个服务器的实现之前对其进行包装。随后从ChannelPipeline读取消息，并将其发送到已实现的服务器实例。

图15-6展示了Finagle的客户端和服务器之间的关系。

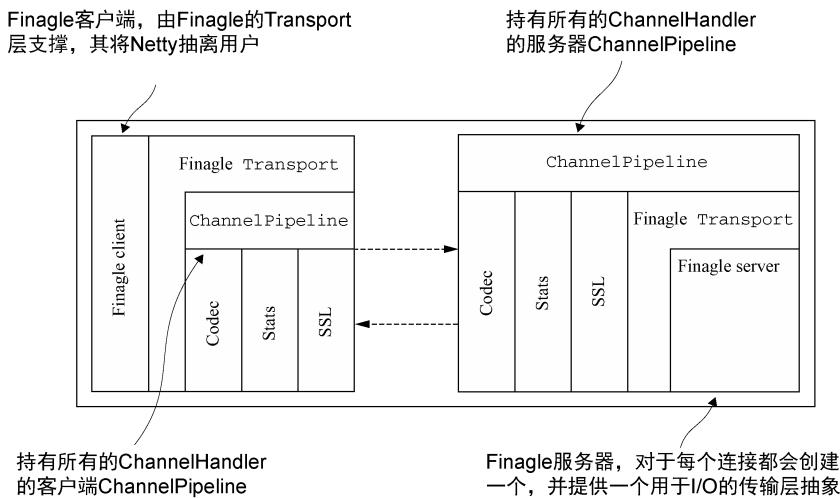


图 15-6 Netty 的使用

### Netty/Finagle 桥接器

代码清单 15-1 展示了一个使用默认选项的静态的 ChannelFactory。

#### 代码清单 15-1 设置 ChannelFactory

```
object Netty3Transporter {
    val channelFactory: ChannelFactory =
        new NioClientSocketChannelFactory(
            Executors, 1 /*# boss threads*/, WorkerPool, DefaultTimer
        )
    {
        // no-op; unreleasable
        override def releaseExternalResources() = {}
    }
    val defaultChannelOptions: Map[String, Object] = Map(
        "tcpNoDelay" -> java.lang.Boolean.TRUE,
        "reuseAddress" -> java.lang.Boolean.TRUE
    )
}
```

↑ 创建一个 ChannelFactory 的实例

↑ 设置用于新 Channel 的选项

<sup>①</sup> 相关的类可以在 <https://github.com/twitter/finagle/blob/develop/finagle-netty4/src/main/scala/com/twitter/finagle/netty4/transport/ChannelTransport.scala> 找到。——译者注

这个 ChannelFactory 桥接了 Netty 的 Channel 和 Finagle 的 Transport( 为简洁起见，这里移除了统计代码 )。当通过 apply 方法被调用时，这将创建一个新的 Channel 以及 Transport。当该 Channel 已经连接或者连接失败时，将会返回一个被完整填充的 Future。

代码清单 15-2 展示了将 Channel 连接到远程主机的 ChannelConnector。

### 代码清单 15-2 连接到远程服务器

```
private[netty3] class ChannelConnector[In, Out] {
    newChannel: () => Channel,
    newTransport: Channel => Transport[In, Out]
} extends (SocketAddress => Future[Transport[In, Out]]) {
    def apply(addr: SocketAddress): Future[Transport[In, Out]] = {
        require(addr != null)
        val ch = try newChannel() catch {
            case NonFatal(exc) => return Future.exception(exc)
        }
        // Transport is now bound to the channel; this is done prior to
        // it being connected so we don't lose any messages.
        val transport = newTransport(ch)
        val connectFuture = ch.connect(addr)
        val promise = new Promise[Transport[In, Out]]
        promise.setInterruptHandler { case _cause =>
            // Propagate cancellations onto the netty future.
            connectFuture.cancel()
        }
        connectFuture.addListener(new ChannelFutureListener {
            def operationComplete(f: ChannelFuture) {
                if (f.isSuccess) {
                    promise.setValue(transport)
                } else if (f.isCancelled) {
                    promise.setException(
                        WriteException(new CancelledConnectionException))
                } else {
                    promise.setException(WriteException(f.getCause))
                }
            }
        })
        promise.onFailure { _ => Channels.close(ch)
        }
    }
}
```

这个工厂提供了一个 ChannelPipelineFactory，它是一个 Channel 和 Transport 的工厂。该工厂是通过 apply 方法调用的。一旦被调用，就会创建一个新的 ChannelPipeline ( newPipeline )。ChannelFactory 将会使用这个 ChannelPipeline 来创建新的 Channel，随后使用所提供的选项 ( newConfiguredChannel ) 对它进行配置。配置好的 Channel 将会被作为一个匿名的工厂传递给一个 ChannelConnector。该连接器将会被调用，并返回一个 Future[Transport]。

代码清单 15-3 展示了细节<sup>①</sup>。

### 代码清单 15-3 基于 Netty 3 的传输

```

case class Netty3Transporter[In, Out](
    pipelineFactory: ChannelPipelineFactory,
    newChannel: ChannelPipeline => Channel =
        Netty3Transporter.channelFactory.newChannel(_),
    newTransport: Channel => Transport[In, Out] =
        new ChannelTransport[In, Out](_),
    // various timeout/ssl options
) extends (
    (SocketAddress, StatsReceiver) => Future[Transport[In, Out]]
) {
    private def newPipeline(
        addr: SocketAddress,
        statsReceiver: StatsReceiver
    ) = {
        val pipeline = pipelineFactory.getPipeline()           | 创建一个 ChannelPipeline，并添
        // add stats, timeouts, and ssl handlers             | 加所需的 ChannelHandler
        pipeline
    }
    private def newConfiguredChannel(
        addr: SocketAddress,
        statsReceiver: StatsReceiver
    ) = {
        val ch = newChannel(newPipeline(addr, statsReceiver))
        ch.getConfig.setOptions(channelOptions.asJava)
        ch
    }
    def apply(
        addr: SocketAddress,
        statsReceiver: StatsReceiver
    ): Future[Transport[In, Out]] = {
        val conn = new ChannelConnector[In, Out](
            () => newConfiguredChannel(addr, statsReceiver),
            newTransport, statsReceiver)                      | 创建一个内部使用的
        conn(addr)                                         | ChannelConnector
    }
}

```

Finagle 服务器使用 Listener 将自身绑定到给定的地址。在这个示例中，监听器提供了一个 ChannelPipelineFactory、一个 ChannelFactory 以及各种选项（为了简洁起见，这里没包括）。我们使用一个要绑定的地址以及一个用于通信的 Transport 调用了 Listener。接着，创建并配置了一个 Netty 的 ServerBootstrap。然后，创建了一个匿名的 ServerBridge 工厂，递给 ChannelPipelineFactory，其将被递交给该引导服务器。最后，该服务器将会被绑定到给定的地址。

现在，让我们来看看基于 Netty 的 Listener 实现，如代码清单 15-4 所示。

<sup>①</sup> Finagle 的源代码位于 <https://github.com/twitter/finagle>。

## 代码清单 15-4 基于 Netty 的 Listener 实现

```

case class Netty3Listener[In, Out](
    pipelineFactory: ChannelPipelineFactory,
    channelFactory: ServerChannelFactory
    bootstrapOptions: Map[String, Object], ... // stats/timeouts/ssl config
) extends Listener[In, Out] {
    def newServerPipelineFactory(
        statsReceiver: StatsReceiver, newBridge: () => ChannelHandler
    ) = new ChannelPipelineFactory {                                ←
        def getPipeline() = {                                     ←
            val pipeline = pipelineFactory.getPipeline()
            ... // add stats/timeouts/ssl
            pipeline.addLast("finagleBridge", newBridge())      ←
            pipeline                                         ←
        }
    }
    def listen(addr: SocketAddress) {
        serveTransport: Transport[In, Out] => Unit
    }: ListeningServer =
        new ListeningServer with CloseAwaitably {
            val newBridge = () => new ServerBridge(serveTransport, ...)
            val bootstrap = new ServerBootstrap(channelFactory)
            bootstrap.setOptions(bootstrapOptions.asJava)
            bootstrap.setPipelineFactory(
                newServerPipelineFactory(scopedStatsReceiver, newBridge))
            val ch = bootstrap.bind(addr)
        }
    }
}

```

当一个新的Channel打开时，该桥接器将会创建一个新的ChannelTransport并将其递回给Finagle服务器。代码清单 15-5 展示了所需的代码<sup>①</sup>。

## 代码清单 15-5 桥接 Netty 和 Finagle

```

class ServerBridge[In, Out] {
    serveTransport: Transport[In, Out] => Unit,
) extends SimpleChannelHandler {
    override def channelOpen(
        ctx: ChannelHandlerContext,
        e: ChannelStateEvent
    ) {
        val channel = e.getChannel
        val transport = new ChannelTransport[In, Out](channel)
        serveTransport(transport)
        super.channelOpen(ctx, e)
    }
    override def exceptionCaught(

```

<sup>①</sup> 完整的源代码在 <https://github.com/twitter/finagle>。

```

    ctx: ChannelHandlerContext,
    e: ExceptionEvent
) { // log exception and close channel
}
}

```

### 15.2.4 Finagle 的抽象

Finagle的核心概念是一个从Request到Future[Response]<sup>①</sup>的简单函数（函数式编程语言是这里的关键）。

```
type Service[Req, Rep] = Req => Future[Rep]②
```

这种简单性释放了非常强大的组合性。Service 是一种对称的 API，同时代表了客户端以及服务器。服务器实现了该服务的接口。该服务器可以被具体地用于测试，或者 Finagle 也可以将它暴露到网络接口上。客户端将被提供一个服务实现，其要么是虚拟的，要么是某个远程服务器的具体表示。

例如，我们可以通过实现一个服务来创建一个简单的 HTTP 服务器，该服务接受 HttpReq 作为参数，返回一个代表最终响应的 Future[HttpRep]。

```

val s: Service[HttpReq, HttpRep] = new Service[HttpReq, HttpRep] {
    def apply(req: HttpReq): Future[HttpRep] =
        Future.value(HttpRep(Status.OK, req.body))
}
Http.serve(":80", s)

```

随后，客户端将被提供一个该服务的对称表示。

```

val client: Service[HttpReq, HttpRep] = Http.newService("twitter.com:80")
val f: Future[HttpRep] = client(HttpReq("/"))
f map { rep => processResponse(rep) }

```

这个例子将把该服务器暴露到所有网络接口的 80 端口上，并从 twitter.com 的 80 端口消费。我们也可以选择不暴露该服务器，而是直接使用它。

```
server(HttpReq("/")) map { rep => processResponse(rep) }
```

在这里，客户端代码有相同的行为，只是不需要网络连接。这使得测试客户端和服务器非常简单直接。

客户端以及服务器都提供了特定于应用程序的功能。但是，也有对和应用程序无关的功能的需求。这样的例子如超时、身份验证以及统计等。Filter 为实现应用程序无关的功能提供了抽象。

过滤器接收一个请求和一个将被它组合的服务：

```
type Filter[Req, Rep] = (Req, Service[Req, Rep]) => Future[Rep]
```

<sup>①</sup> 这里的 Future[Response] 相当于 Java 8 中的 CompletionStage<Response>。——译者注

<sup>②</sup> 虽然不完全等价，但是可以理解为 Java 8 的 public interface Service<Req, Rep> extends Function<Req, CompletionStage<Rep>>{}。——译者注

多个过滤器可以在被应用到某个服务之前链接在一起：

```
recordHandleTime andThen
traceRequest andThen
collectJvmStats andThen
myService
```

这允许了清晰的逻辑抽象以及良好的关注点分离。在内部，Finagle 大量地使用了过滤器，其有助于提高模块化以及可复用性。它们已经被证明，在测试中很有价值，因为它们通过很小的模拟便可以被独立地单元测试。

过滤器可以同时修改请求和响应的数据以及类型。图 15-7 展示了一个请求，它在通过一个过滤器链之后到达了某个服务并返回。

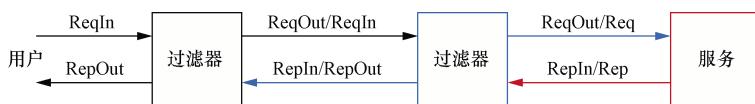


图 15-7 请求/响应流

我们可以使用类型修改来实现身份验证。

```
val auth: Filter[HttpReq, AuthHttpReq, HttpRes, HttpRes] =
  { (req, svc) => authReq(req).flatMap { authReq => svc(authReq) } }

val authedService: Service[AuthHttpReq, HttpRes] = ...
val service: Service[HttpReq, HttpRes] =
  auth andThen authedService
```

这里我们有一个需要 AuthHttpReq 的服务。为了满足这个需求，创建了一个能接收 HttpReq 并对它进行身份验证的过滤器。随后，该过滤器将和该服务进行组合，产生一个新的可以接受 HttpReq 并产生 HttpRes 的服务。这使得我们可以从该服务隔离，单独地测试身份验证过滤器。

### 15.2.5 故障管理

我们假设故障总是会发生；硬件会失效、网络会变得拥塞、网络链接会断开。对于库来说，如果它们正在上面运行的或者正在与之通信的系统发生故障，那么库所拥有的极高的吞吐量以及极低的延迟都将毫无意义。为此，Finagle 是建立在有原则地管理故障的基础之上的。为了能够更好地管理故障，它牺牲了一些吞吐量以及延迟。

Finagle 可以通过隐式地使用延迟作为启发式(算法的因子)来均衡跨集群主机的负载。Finagle 客户端将在本地通过统计派发到单个主机的还未完成的请求数来追踪它所知道的每个主机上的负载。有了这些信息，Finagle 会将新的请求(隐式地)派发给具有最低负载、最低延迟的主机。

失败的请求将导致 Finagle 关闭到故障主机的连接，并将它从负载均衡器中移除。在后台，Finagle 将不断地尝试重新连接。只有在 Finagle 能够重新建立一个连接时，该主机才会被重新加入到负载均衡器中。然后，服务的所有者可以自由地关闭各个主机，而不会对下游的客户端造成

负面影响。

### 15.2.6 组合服务

Finagle 的服务即函数（service-as-a-function）的观点允许编写简单但富有表现力的代码。例如，一个用户发出的对于他们的主页时间线的请求涉及了大量的服务，其中的核心是身份验证服务、时间线服务以及推特服务。这些关系可以被简洁地表达。

代码清单 15-6 通过 Finagle 组合服务

```
val timelineSvc = Thrift.newIface[TimelineService](...)
val tweetSvc = Thrift.newIface[TweetService](...)
val authSvc = Thrift.newIface[AuthService](...)           ↗为每个服务创建一个客户端

val authFilter = Filter.mk[Req, AuthReq, Res, Res] { (req, svc) =>
    authSvc.authenticate(req) flatMap svc(_)
}

val apiService = Service.mk[AuthReq, Res] { req =>
    timelineSvc(req.userId) flatMap { tl =>
        val tweets = tl map tweetSvc.getById(_)
        Future.collect(tweets) map tweetsToJson(_)
    }
}

Http.serve(":80", authFilter andThen apiService)           ↗使用该身份验证过滤器以及
                                                               我们的服务在 80 端口上启动
                                                               一个新的 HTTP 服务
```

在这里，我们为时间线服务、推特服务以及身份验证服务都创建了客户端。并且，为了对原始的请求进行身份验证，创建了一个过滤器。最后，我们实现的服务，结合了身份验证过滤器，暴露在 80 端口上。

当收到请求时，身份验证过滤器将尝试对它进行身份验证。错误都会被立即返回，不会影响核心业务。身份验证成功之后，AuthReq 将会被发送到 API 服务。该服务将会使用附加的 userId 通过时间线服务来查找该用户的时间线。然后，返回一组推特 ID，并在稍后遍历。每个 ID 都会被用来请求与之相关联的推特。最后，这组推特请求会被收集起来，转换为一个 JSON 格式的响应。

正如你所看到的，我们定义了数据流，并且将并发的问题留给了 Finagle。我们不必管理线程池，也不必担心竞态条件。这段代码既清晰又安全。

### 15.2.7 未来：Netty

为了改善Netty的各个部分，让Finagle以及更加广泛的社区都能够从中受益，我们一直在与Netty的维护者密切合作<sup>①</sup>。最近，Finagle的内部结构已经升级为更加模块化的结构，为升级到

<sup>①</sup> “Netty 4 at Twitter: Reduced GC Overhead”：<https://blog.twitter.com/2013/netty-4-at-twitter-reduced-gc-overhead>。

Netty 4 铺平了道路。

### 15.2.8 Twitter 小结

Finagle已经取得了辉煌的成绩。我们已经想方设法大幅度地提高了我们所能够处理的流量，同时也降低了延迟以及硬件需求。例如，在将我们的API端点从Ruby技术栈迁移到Finagle之后，我们看到，延迟从数百毫秒下降到了数十毫秒之内，同时还将所需要的机器数量从3位数减少到了个位数。我们新的技术栈已经使得我们达到了新的吞吐量记录。在撰写本文时，我们所记录的每秒的推特数是 143 199<sup>①</sup>。这一数字对于我们的旧架构来说简直是难以想象的。

Finagle 的诞生是为了满足 Twitter 横向扩展以支持全球数以亿计的用户的需求，而在当时支撑数以百万计的用户并保证服务在线已然是一项艰巨的任务了。使用 Netty 作为基础，我们能够快速地设计和建造 Finagle，以解决我们的伸缩性难题。Finagle 和 Netty 处理了 Twitter 所遇到的每一个请求。

## 15.3 小结

本章深入了解了对于像 Facebook 以及 Twitter 这样的大公司是如何使用 Netty 来保证最高水准的性能以及灵活性的。

- Facebook 的 Nifty 项目展示了，如何通过提供自定义的协议编码器以及解码器，利用 Netty 来替换现有的 Thrift 实现。
- Twitter 的 Finagle 展示了，如何基于 Netty 来构建你自己的高性能框架，并通过类似于负载均衡以及故障转移这样的特性来增强它的。

我们希望这里所提供的案例研究，能够成为你打造下一代杰作的时候的信息和灵感的来源。

---

<sup>①</sup> “New Tweets per second record, and how!”：<https://blog.twitter.com/2013/new-tweets-per-second-recordand-how>。

# 附录 Maven 介绍

---

本附录提供了对 Apache Maven (<http://maven.apache.org/what-is-maven.html>) 的基本介绍。在读过之后，你应该能够通过复用本书示例中的配置来启动你自己的项目。

Maven 是一个强大的工具，学习的回报很大。如果希望了解更多，你可以在 <http://maven.apache.org> 找到官方文档，在 [www.sonatype.com/resources/books](http://www.sonatype.com/resources/books) 找到一套极好的免费的 PDF 格式的书。

第一节将介绍 Maven 的基本概念。在第二节中，我们将使用本书示例项目中的示例来说明这些基本概念。

## A.1 什么是 Maven

Maven 是一种用来管理 Java 项目的工具，但不是那种用来管理资源规划和调度的工具。相反，它处理的是管理一个具体的项目所涉及的各种任务，如编译、测试、打包、文档以及分发。

Maven 包括以下的几个部分。

- 一组用于处理依赖管理、目录结构以及构建工作流的约定。基于这些约定实现的标准化可以极大地简化开发过程。例如，一个常用的目录结构使得开发者可以更加容易地跟上不熟悉的项目的节奏。
- 一个用于项目配置的 XML Schema：项目对象模型（Project Object Model），简称 POM<sup>①</sup>。每一个 Maven 项目都拥有一个 POM 文件<sup>②</sup>，默认命名为 pom.xml，包含了 Maven 用于管理该项目的所有配置信息。
- 一个委托外部组件来执行项目任务的插件架构。这简化了更新以及扩展 Maven 能力的过程。

---

① Maven 项目，“What is a POM？”：<http://maven.apache.org/guides/introduction/introduction-to-the-pom.html>。

② 在 <http://maven.apache.org/ref/3.3.9/maven-model/maven.html> 有关于 POM 的详细描述。

构建和测试我们的示例项目只需要用到 Maven 多种特性的一个子集。这些也是我们将在本附录中所讨论的内容，其中不包括那些在生产部署中肯定需要用到的特性。我们将会涵盖的主题包括以下内容。

- 基本概念：构件、坐标以及依赖。
- 关键元素以及 Maven 项目描述符（pom.xml）的用法。
- Maven 构建的生命周期以及插件。

## A.1.1 安装和配置 Maven

可以从<http://maven.apache.org/download.cgi>下载适合于你的系统的Maven tar.gz或者zip文件。安装非常简单：将该归档内容解压到你选择的任意文件夹（我们称之为<安装目录>）中。这将创建目录<安装目录>\apache-maven-3.3.9<sup>①</sup>。

然后，

- 将环境变量 M2\_HOME 设置为指向<安装目录>\apache-maven-3.3.9，这个环境变量将会告诉 Maven 在哪里能找到它的配置文件，conf\settings.xml；
- 将%M2\_HOME%\bin（在 Linux 上是\${M2\_HOME}/bin）添加到你的执行路径，在这之后，在命令行执行 mvn 就能运行 Maven 了。

在编译和运行示例项目时，不需要修改默认配置。在首次执行mvn时，Maven会为你创建本地存储库<sup>②</sup>，并从Maven中央存储库下载基本操作所需的大量JAR文件。最后，它会下载构建当前项目所需要的依赖项（包括Netty的JAR包）。关于自定义settings.xml的详细信息可以在<http://maven.apache.org/settings.html>找到。

## A.1.2 Maven 的基本概念

在下面的章节中，我们将解释 Maven 的几个最重要的概念。熟悉这些概念将有助于你理解 POM 文件的各个主要元素。

### 1. 标准的目录结构

Maven定义了一个标准的项目目录结构<sup>③</sup>。并不是每种类型的项目都需要Maven的所有元素，很多都可以在必要的时候在POM文件中重写。表A-1 展示了一个基本的WAR项目，有别于JAR项目，它拥有src/main/webapp文件夹。当Maven构建该项目时，该目录（其中包含WEB-INF目录）的内容将会被放置在WAR文件的根路径上。位于该文件树根部的\${project.basedir}

① 在本书中文版出版时，Maven 的最新版本是 3.3.9。

② 在默认情况下，这是你当前操作系统的 HOME 目录下的.m2/repository 目录。

③ 有关标准目录结构的优点，参考 <http://maven.apache.org/guides/introduction/introductionto-the-standard-directory-layout.html>。

是一个标准的Maven属性，标识了当前项目的根目录。

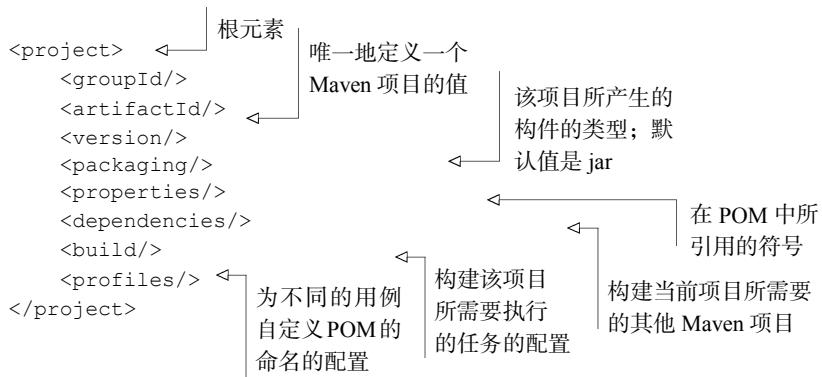
表 A-1 基本的项目目录结构

文件夹	描述
\\${project.basedir}	项目根路径
---\src	源代码根路径
---\main	程序代码
---\java	Java 源代码
---\resources	属性文件、XML schema 等
---\webapp	Web 应用程序资源
---\test	测试源代码根路径
---\java	Java 源代码，如 JUnit 测试类
---\resources	属性文件、XML schema 等
---\target	由构建过程所创建的文件

## 2. POM 大纲

代码清单 A-1 是我们的一个示例项目的 POM 文件的大纲。只显示了顶层的 schema 元素。其中的一些也是其他元素的容器。

代码清单 A-1 POM 文件的大纲



我们将在本节剩下的部分中更加详细地讨论这些元素。

## 3. 构件

任何可以被 Maven 的坐标系统（参见接下来的关于 GAV 坐标的讨论）唯一标识的对象都是

一个 Maven 构件。大多数情况下，构件是构建 Maven 项目所生成的文件，如 JAR。但是，只包含其他 POM（该文件本身并不产生构件）使用的定义的 POM 文件也是 Maven 构件。

Maven 构件的类型由其 POM 文件的<packaging>元素指定。最常用的值是 pom、jar、ear、war 以及 maven-plugin。

#### 4. POM 文件的用例

可以通过以下的方式来使用 POM 文件。

- 默认的——用于构建一个构件。
- 父 POM——提供一个由子项目继承的单个配置信息源——声明这个 POM 文件作为它们的<parent>元素的值。
- 聚合器——用于构建一组声明为<modules>的项目，这些子项目位于其当前聚合器项目的文件夹中，每个都包含有它自己的 POM 文件。

作为父 POM 或者聚合器的 POM 文件的<packaging>元素的值将是 pom。注意，一个 POM 文件可能同时提供两项功能。

#### 5. GAV 坐标

POM 定义了 5 种称为坐标的元素，用于标识 Maven 构件。首字母缩写 GAV 指的是必须始终指定的 3 个坐标<groupId>、<artifactId>以及<version>的首字母。

下面的坐标是按照它们在坐标表达式中出现的顺序列出的。

(1) <groupId>是项目或者项目组的全局的唯一标识符。这通常是 Java 源代码中使用的全限定的 Java 包名。例如，io.netty、com.google。

(2) <artifactId>用于标识和某个<groupId>相关的不同的构件。例如，netty-all、netty-handler。

(3) <type>是指和项目相关的主要构件的类型（对应于构件的 POM 文件中的<packaging>值）。它的默认值是 jar。例如，pom、war、ear。

(4) <version>标识了构件的版本。例如，1.1、2.0-SNAPSHOT<sup>①</sup>、4.1.9.Final。

(5) <classifier>用于区分属于相同的 POM 但是却被以不同的方式构建的构件。例如，javadoc、sources、jdk16、jdk17。

一个完整的坐标表达式具有如下格式：

```
artifactId:groupId:packaging:version:classifier
```

下面的 GAV 坐标标识了包含所有 Netty 组件的 JAR：

```
io.netty:netty-all:4.1.9.Final
```

POM 文件必须声明它所管理的构件的坐标。一个具有如下坐标的项目：

---

<sup>①</sup> 有关 SNAPSHOT 构件的更多信息参见本节后面关于“快照和发布”的讨论。

```
<groupId>io.netty</groupId>
<artifactId>netty-all</artifactId>
<version>4.1.9.Final</version>
<packaging>jar</packaging>
```

将会产生一个具有以下格式的名称的构件：

```
<artifactId>-<version>.<packaging>
```

在这种情况下，它将产生这个构件：

```
netty-all-4.1.9.Final.jar
```

## 6. 依赖

项目的依赖是指编译和执行它所需要的外部构件。在大多数情况下，你的项目的依赖项也会有它自己的依赖。我们称这些依赖为你的项目的传递依赖。一个复杂的项目可能会有一个深层级的依赖树；Maven提供了各种用于帮助理解和管理它的工具。<sup>①</sup>

Maven的`<dependency>`<sup>②</sup>声明在POM的`<dependencies>`元素中：

```
<dependencies>
    <dependency>
        <groupId/>
        <artifactId/>
        <version/>
        <type/>
        <scope/>
        <systemPath/>
    </dependency>
    ...
</dependencies>
```

在`<dependency>`声明中，GAV坐标总是必不可少的<sup>③</sup>。`type`以及`scope`元素对于那些值不分别是默认值`jar`和`compile`的依赖来说也是必需的。

下面的代码示例是从我们示例项目的顶级 POM 中摘录的。注意第一个条目，它声明了对我们先前提到的 Netty JAR 的依赖。

```
<dependencies>
    <dependency>
        <groupId>io.netty</groupId>
        <artifactId>netty-all</artifactId>
        <version>4.1.9.Final</version>
    </dependency>
    <dependency>
        <groupId>nia</groupId>
```

① 例如，在拥有 POM 文件的项目目录中，在命令行执行“mvn dependency:tree”。

② 管理依赖项：<http://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html>。

③ 参见后面的“依赖管理”小节。

```
<artifactId>util</artifactId>
<version>1.0-SNAPSHOT</version>
</dependency>
<dependency>
    <groupId>com.google.protobuf</groupId>
    <artifactId>protobuf-java</artifactId>
    <version>2.5.0</version>
</dependency>
<dependency>
    <groupId>org.eclipse.jetty.npn</groupId>
    <artifactId>npn-api</artifactId>=
    <version>1.1.0.v20120525</version>
</dependency>
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
</dependency>
</dependencies>
```

<scope>元素可以具有以下值。

- `compile`——编译和执行需要的（默认值）。
- `runtime`——只有执行需要。
- `optional`——不被引用了这个项目所产生的构件的其他项目，视为传递依赖。
- `provided`——不会被包含在由这个 POM 产生的 WAR 文件的 WEB\_INF/lib 目录中。
- `test`——只有编译和测试的执行需要。
- `import`——这将在后面的“依赖管理”一节进行讨论。

<systemPath>元素用来指定文件系统中的绝对位置。

Maven 用来管理项目依赖的方式，包括了一个用来存储和获取这些依赖的存储库协议，已经彻底地改变了在项目之间共享 JAR 文件的方式，从而有效地消除了项目的中每个开发人员都维护一个私有 lib 目录时经常会出现的问题。

## 7. 依赖管理

POM 的<dependencyManagement>元素包含可以被其他项目使用的<dependency>声明。这样的 POM 的子项目将会自动继承这些声明。其他项目可以通过使用<scope>元素的 `import` 值来导入它们（将在稍后讨论）。

引用了<dependencyManagement>元素的项目可以使用它所声明的依赖，而不需要指定它们的<version>坐标。如果<dependencyManagement>中的<version>在稍后有所改变，则它将被所有引用它的 POM 拾起。

在下面的示例中，所使用的 Netty 版本是在 POM 的<properties>部分中定义，在<dependencyManagement>中引用的。

```
<properties>
    <netty.version>4.1.9</netty.version>
    ...
    ...
</properties>
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>io.netty</groupId>
            <artifactId>netty-all</artifactId>
            <version>${netty.version}</version>
        </dependency>
    </dependencies>
    ...
</dependencyManagement>
```

对于这种使用场景，依赖的`<scope>`元素有一个特殊的`import`值：它将把外部 POM（没有被声明为`<parent>`）的`<dependencyManagement>`元素的内容导入到当前 POM 的`<dependencyManagement>`元素中。

## 8. 构建的生命周期

Maven 构建的生命周期是一个明确定义的用于构建和分发构件的过程。有 3 个内置的构建生命周期：`clean`、`default` 和 `site`。我们将只讨论其中的前两个，分别用于清理和分发项目。

一个构建的生命周期由一系列的阶段所组成。下面是默认的构建生命周期的各个阶段的一个部分清单。

- `validate`——检查项目是否正确，所有必需的信息是否已经就绪。
- `process-sources`——处理源代码，如过滤任何值。
- `compile`——编译项目的源代码。
- `process-test-resources`——复制并处理资源到测试目标目录中。
- `test-compile`——将测试源代码编译到测试目标目录中。
- `test`——使用合适的单元测试框架测试编译的源代码。
- `package`——将编译的代码打包为它的可分发格式，如 JAR。
- `integration-test`——处理并将软件包部署到一个可以运行集成测试的环境中。
- `verify`——运行任何的检查以验证软件包是否有效，并且符合质量标准。
- `install`——将软件包安装到本地存储库中，在那里其他本地构建项目可以将它引用为依赖。
- `deploy`——将最终的构件上传到远程存储库，以与其他开发人员和项目共享。

执行这些阶段中的一个阶段将会调用所有前面的阶段。例如：

```
mvn package
```

将会执行 validate、compile 以及 test，并随后将该构件组装到该项目的目标目录中。

执行

```
mvn clean install
```

将会首先移除所有先前的构建所创建的结果。然后，它将会运行所有到该阶段的默认阶段，并且包括将该构件放置到你的本地存储库的文件系统中。

虽然我们的示例项目可以通过这些简单的命令来构建，但是任何使用Maven的重要工作都需要详细了解构建生命周期的各个阶段。<sup>①</sup>

## 9. 插件

虽然Maven协调了所有构建生命周期阶段的执行，但是它并没有直接实现它们，相反，它将它们委托给了插件<sup>②</sup>，这些插件是maven-plugin类型的构件（打包为JAR文件）。Apache Maven项目为标准构建生命周期所定义的所有任务都提供了插件，更多的是由第三方生产的，用于处理各种自定义的任务。

插件可能拥有多个内部步骤，或者目标，其也可以被单独调用。例如，在一个 JAR 项目中，默认的构建生命周期由 maven-jar-plugin 处理，其将构建的各个阶段映射到了它自己的以及其他插件的目标中，如表 A-2 所示。

表 A-2 阶段、插件以及目标

阶 段	插件：目标
process-resources	resources:resources
compile	compiler:compiler
process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	jar:jar
install	install:install
deploy	deploy:deploy

在我们的示例项目中，我们使用了下面的第三方插件来从命令行执行我们的项目。注意插件的声明，它被打包为 JAR 包，使用了和<dependency>的 GAV 坐标相同的 GAV 坐标。

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>exec-maven-plugin</artifactId>
```

① “Introduction to the Build Lifecycle”：<http://maven.apache.org/guides/introduction/introduction-to-thelifecycle.html>。

② “Available Plugins”：<http://maven.apache.org/plugins/index.html>。

```
<version>1.2.1</version>
</plugin>
```

## 10. 插件管理

如同`<dependencyManagement>`, `<pluginManagement>`声明了其他 POM 可以使用的信息, 如代码清单 A-2 所示。但是这只适用于子 POM, 因为对于插件来说, 没有导入声明。和依赖一样, `<version>`坐标是继承的。

### 代码清单 A-2 `pluginManagement`

```
<build>
  <pluginManagement>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.2</version>
        <configuration>
          <source>1.7</source>
          <target>1.7</target>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>exec-maven-plugin</artifactId>
        <version>1.2.1</version>
      </plugin>
    </plugins>
  </pluginManagement>
</build>
```

代码清单 A-3 展示了代码清单 A-2 中的 POM 片段的子 POM 是如何使用其父 POM 的`<pluginManagement>`配置的, 它只引用了其构建所需的插件。子 POM 还可以重写它需要自定义的任何插件配置。

### 关于 Maven 插件

在声明由 Maven 项目生成的插件时, 可以省略`groupId` (`org.apache.maven.plugins`), 如代码清单 A-2 中的`maven-compiler-plugin` 的声明中所示。此外, 保留了以“maven”开头的`artifactId`, 仅供 Maven 项目使用。例如, 第三方可以提供一个`artifactId` 为`exec-maven-plugin` 的插件, 但是不能为`maven-exec-plugin`。

POM 定义了一个大多数插件都需要遵从的插件配置格式。

更多的信息参见 Maven 的“插件配置指南” (<http://maven.apache.org/guides/mini/guide-configuring-plugins.html>)。这将帮助你设置你想要在自己的项目中使用的任何插件。

**代码清单 A-3 插件继承**

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
    </plugin>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>exec-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

## 11. 配置文件

配置文件（在`<profiles>`中定义）是一组自定义的 POM 元素，可以通过自动或者手动启用（激活）来改变 POM 的行为。例如，你可以定义一个配置文件，它将根据 JDK 版本、操作系统或者目标部署环境（如开发、测试或者生产环境）来设置构建参数。

可以通过命令行的`-P` 标志来显式地引用配置文件。下面的例子将激活一个将 POM 自定义为使用 JDK1.6 的配置文件。

```
mvn -P jdk16 clean install
```

## 12. 存储库

Maven 的构件存储库<sup>①</sup>可能是远程的，也可能是本地的。

- 远程存储库是一个 Maven 从其下载 POM 文件中所引用的依赖的服务。如果你有上传权限，那么这些依赖中可能也会包含由你自己的项目所产生的构件。大量开放源代码的 Maven 项目（包含 Netty）都将它们的构件发布到可以公开访问的 Maven 存储库。
- 本地存储库是一个本地的目录，其包含从远程存储库下载的构件，以及你在本地机器上构建并安装的构件。它通常放在你的主目录下，如：

```
C:\Users\maw\.m2\repository
```

Maven 存储库的物理目录结构使用 GAV 坐标，如同 Java 编译器使用包名一样。例如，在 Maven 下载了下面的依赖之后：

```
<dependency>
  <groupId>io.netty</groupId>
  <artifactId>netty-all</artifactId>
  <version>4.1.9.Final</version>
</dependency>
```

---

<sup>①</sup> 参考 <http://maven.apache.org/guides/introduction/introduction-to-repositories.html>。

将会在本地存储库中找到以下内容：

```
.m2\repository
|---\io
|---\netty
|---\netty-all
|---\4.1.9.Final
    netty-all-4.1.9.Final.jar
    netty-all-4.1.9.Final.jar.sha1
    netty-all-4.1.9.Final.pom
    netty-all-4.1.9.Final.pom.sha1
    _maven.repositories
```

### 13. 快照和发布

远程存储库通常会为正在开发的构件，以及那些稳定发布或者生产发布的构件，定义不同的区域。这些区域被分别称为快照存储库和发布存储库。

一个`<version>`值由`-SNAPSHOT`结尾的构件将被认为是还没有发布的。这种构件可以重复地使用相同的`<version>`值被上传到存储库。每次它都会被分配一个唯一的时间戳。当项目检索构件时，下载的是最新实例。

一个`<version>`值不具有`-SNAPSHOT`后缀的构件将会被认为是一个发布版本。通常，存储库策略只允某一特定的发布版本上传一次。

当构建一个具有`SNAPSHOT`依赖的项目时，Maven 将检查本地存储库中是否有对应的副本。如果没有，它将尝试从指定的远程存储库中检索，在这种情况下，它将接收到具有最新时间戳的构件。如果本地的确有这个构件，并且当前构建也是这一天中的第一个，那么 Maven 将默认尝试更新该本地副本。这个行为可以通过使用 Maven 配置文件 (`settings.xml`) 中的配置或者命令行标志来进行配置。

## A.2 POM 示例

在这一节中，我们将通过介绍一些 POM 示例来说明我们在前一节中所讨论的主题。

### A.2.1 一个项目的 POM

代码清单 A-4 展示了一个 POM，其为一个简单的 Netty 项目创建了一个 JAR 文件。

**代码清单 A-4 独立的 pom.xml**

```
<?xml version="1.0" encoding="ISO-8859-15"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/maven-v4_0_0.xsd">
```

```

<modelVersion>4.0.0</modelVersion>
<groupId>com.example</groupId>
<artifactId>mypackage</artifactId>
<version>1.0-SNAPSHOT</version>

<packaging>jar</packaging>
<name>My Jar Project</name>

<dependencies>
    <dependency>
        <groupId>io.netty</groupId>
        <artifactId>netty-all</artifactId>
        <version>4.1.9.Final</version>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.2</version>
            <configuration>
                <source>1.7</source>
                <target>1.7</target>
            </configuration>
        </plugin>
    </plugins>
</build>
</project>

```

该项目的 GAV 坐标

该项目产生的构件将是一个 JAR 文件（默认值）

这个 POM 只声明了 Netty JAR 作为依赖；一个典型的 Maven 项目会有许多依赖

<build>部分声明了用于执行构建任务的插件。我们只自定义了编译器插件，对于其他的插件，我们接受默认值

这个 POM 创建的构件将是一个 JAR 文件，其中包含从项目的 Java 源代码编译而来的类。在编译的过程中，被声明为依赖的 Netty JAR 将会被添加到 CLASSPATH 中。

下面是使用这个 POM 时会用到的基本 Maven 命令。

- 在项目的构建目录（“target”）中创建 JAR 文件：

```
mvn package
```

- 将该 JAR 文件存储到本地存储库中：

```
mvn install
```

- 将该 JAR 文件发布到全局存储库中（如果已经定义了一个）：

```
mvn deploy
```

## A.2.2 POM 的继承和聚合

正如我们之前所提到的，有几种使用 POM 的方式。在这里，我们将讨论它作为父 POM 或者聚合器的用法。

## 1. POM 继承

POM 文件可能包含子项目要继承（并可能重写）的信息。

## 2. POM 聚合

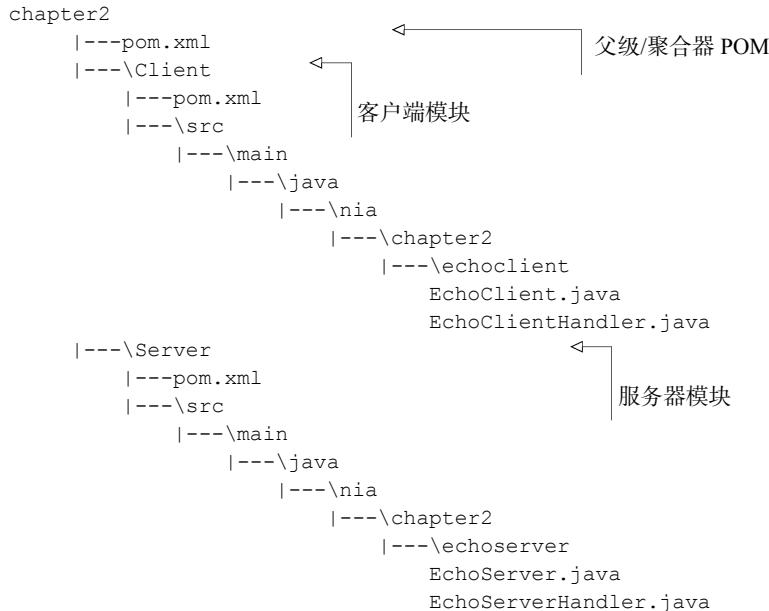
聚合器 POM 会构建一个或者多个子项目，这些子项目驻留在该 POM 所在目录的子目录中。子项目，或者`<modules>`标签，是由它们的目录名标识的：

```
<modules>
  <module>Server</module>
  <module>Client</module>
</modules>
```

当构建子项目时，Maven 将创建一个 *reactor*，它将计算存在于它们之间的任何依赖，以确定它们必须遵照的构建顺序。注意，聚合器 POM 不一定是它声明为模块的项目的父 POM。（每个子项目都可以声明一个不同 POM 作为它的`<parent>`元素的值。）

用于第 2 章的 Echo 客户端/服务器项目的 POM 既是一个父 POM，也是一个聚合器<sup>①</sup>。示例代码根目录下的 chapter2 目录，包含了代码清单 A-5 中所展示的内容。

代码清单 A-5 chapter2 目录树



<sup>①</sup> 它也是它的上级目录中的 nia-samples-parent POM 的一个子 POM，继承了其`<dependencyManagement>`元素的值，并传递给了它自己的子项目。

代码清单 A-6 所示的根级 POM 的打包类型是<pom>，这表示它本身并不产生构件。相反，它会为将它声明为<parent>的项目提供配置信息，如该 Client 和 Server 项目。它也是一个聚合器，这意味着你可以通过在 chapter2 目录中运行 mvn install 来构建它的<modules>中所定义的模块。

### 代码清单 A-6 父级和聚合器 POM: echo-parent

```

<project>
    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>nia</groupId>
        <artifactId>nia-samples-parent</artifactId>
        <version>1.0-SNAPSHOT</version>
    </parent>

    <artifactId>chapter2</artifactId>
    <packaging>pom</packaging>
    <name>2. Echo Client and Server</name>

    <modules>
        <module>Client</module>
        <module>Server</module>
    </modules>

    <properties>
        <echo-server.hostname>localhost</echo-server.hostname>
        <echo-server.port>9999</echo-server.port>
    </properties>

    <dependencies>
        <dependency>
            <groupId>io.netty</groupId>
            <artifactId>netty-all</artifactId>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
            </plugin>
            <plugin>
                <artifactId>maven-failsafe-plugin</artifactId>
            </plugin>
            <plugin>
                <artifactId>maven-surefire-plugin</artifactId>
            </plugin>
            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>exec-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

```

<parent>声明了 samples-parent POM 作为这个 POM 的父 POM

<modules>声明了父 POM 下的目录，其中包含将由这个 POM 来构建的 Maven 项目

<property>值可以通过在命令行上使用 Java 系统属性 (-D) 进行重写。属性由子项目继承

父 POM 的<dependencies>元素由子项目继承

父 POM 的<plugins>元素由子项目继承

```

        </plugins>
    </build>
</project>

```

得益于 Maven 对于继承的支持，该 Server 和 Client 项目的 POM 并没有太多的事情要做。代码清单 A-7 展示了该 Server 项目的 POM。(该 Client 项目的 POM 几乎是完全相同的。)

#### 代码清单 A-7 Echo-服务器项目的 POM

```

<project>
    <parent>
        <groupId>nia</groupId>
        <artifactId>chapter2</artifactId>
        <version>1.0-SNAPSHOT</version>
    </parent>

    <artifactId>echo-server</artifactId>           <parent>声明了父 POM

    <build>
        <plugins>
            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>exec-maven-plugin</artifactId>
                <executions>
                    <execution>
                        <id>run-server</id>
                        <goals>
                            <goal>java</goal>
                        </goals>
                    </execution>
                </executions>
                <configuration>
                    <mainClass>nia.echo.EchoServer</mainClass>
                    <arguments>
                        <argument>${echo-server.port}</argument>
                    </arguments>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>

```

<artifactId>必须声明，因为对于孩子项目来说它是唯一的。<groupId>和<version>，如果没有被定义则从父 POM 继承

exec-maven-plugin 插件可以执行 Maven 命令行的任意命令；在这里，我们用它来运行 Echo 服务器

这个 POM 非常简短，因为它从它的父 POM 和祖父 POM(甚至还还有一个曾祖父级别的 POM 存在——Maven Super POM)那里继承了非常多的信息。注意，例如， `${echo-server.port}` 属性的使用，其继承自父 POM。

Maven 执行的 POM，在组装了所有的继承信息并应用了所有的活动配置文件之后，被称为“有效 POM”。要查看它，请在任何 POM 文件所在的目录中执行下面的 Maven 命令：

```
mvn help:effective-pom
```

## A.3 Maven 命令行

`mvn` 命令的语法如下：

```
mvn [options] [<goal(s)>] [<phase(s)>]
```

有关其用法的详细信息，以及有关我们在这个附录中所讨论的许多主题的更多信息，参见 Sonatype 的《Maven: The Complete Reference》，这是一个很好的资源。<sup>①</sup>

表 A-3 展示了 `mvn` 的命令行选项，这些选项可以通过执行。

```
mvn -help
```

来显示。

表 A-3 `mvn` 的命令行参数

选 项	描 述
<code>-am, --also-make</code>	如果指定了项目列表，还会构建列表所需的项目
<code>-amd, --also-make-dependents</code>	如果指定了项目列表，还会构建依赖于列表中的项目的项目
<code>-B, --batch-mode</code>	在非交互（批处理）模式下运行
<code>-b, --builder &lt;arg&gt;</code>	要使用的构建策略的 id
<code>-C, --strict-checksums</code>	如果校验和不匹配，则让这次构建失败
<code>-c, --lax-checksums</code>	如果校验和不匹配，则发出警告
<code>-cpu, --check-plugin-updates</code>	无效，只是为了保持向后的兼容性
<code>-D, --define &lt;arg&gt;</code>	定义一个系统属性
<code>-e, --errors</code>	生成执行错误的信息
<code>-emp, --encrypt-master-password &lt;arg&gt;</code>	加密主安全密码
<code>-ep, --encrypt-password &lt;arg&gt;</code>	加密服务器密码
<code>-f, --file &lt;arg&gt;</code>	强制使用备用的 POM 文件（或者包含 pom.xml 的目录）
<code>-fae, --fail-at-end</code>	只在最后让构建失败，允许所有不受影响的构建继续进行
<code>-ff, --fail-fast</code>	在反应化的构建中，首次失败便停止构建
<code>-fn, --fail-never</code>	不管项目的结果如何，都决不让构建失败
<code>-gs, --global-settings &lt;arg&gt;</code>	全局设置文件的备用路径
<code>-h, --help</code>	显示帮助信息
<code>-l, --log-file &lt;arg&gt;</code>	所有构建输出的日志文件的位置

<sup>①</sup> 参见 <http://books.sonatype.com/mvnref-book/pdf/mvnref-pdf.pdf>。

续表

选 项	描 述
<code>-llr,--legacy-local-repository</code>	使用 Maven2 的遗留本地存储库 (Legacy Local Repository) 行为; 也就是说, 不使用 <code>_remote.repositories</code> 。也可以通过使用 <code>-Dmaven.legacyLocalRepo=true</code> 激活
<code>-N,--non-recursive</code>	不递归到子项目中
<code>-npr,--no-plugin-registry</code>	无效, 只是为了保持向后的兼容性
<code>-npu,--no-plugin-updates</code>	无效, 只是为了保持向后的兼容性
<code>-nsu,--no-snapshot-updates</code>	取消快照更新
<code>-o,--offline</code>	脱机工作
<code>-P,--activate-profiles &lt;arg&gt;</code>	等待被激活的由逗号分隔的配置文件列表
<code>-pl,--projects &lt;arg&gt;</code>	构建由逗号分隔的指定的 reactor 项目, 而不是所有项目。项目可以通过 [groupId]:artifactId 或者它的相对路径来指定
<code>-q,--quiet</code>	静默输出, 只显示错误
<code>-rf,--resume-from &lt;arg&gt;</code>	从指定的项目恢复 reactor
<code>-s,--settings &lt;arg&gt;</code>	用户配置文件的备用路径
<code>-T,--threads &lt;arg&gt;</code>	线程数目, 如 2.0C, 其中 C 是乘上的 CPU 核心数
<code>-t,--toolchains &lt;arg&gt;</code>	用户工具链文件的备用路径
<code>-U,--update-snapshots</code>	强制检查缺少的发布, 并更新远程存储库上的快照
<code>-up,--update-plugins</code>	无效, 只是为了保持向后的兼容性
<code>-V,--show-version</code>	显示版本信息而不停止构建
<code>-v,--version</code>	显示版本信息
<code>--debug</code>	生成执行调试输出

## A.4 小结

在本附录中, 我们介绍了 Apache Maven, 涵盖了它的基本概念和主要的用例。我们通过本书示例项目中的例子说明了这一切。

我们目标是帮助你更好地理解这些项目的构建方式, 并为独立开发提供了一个起点。

# Netty 实战

Netty 是一款基于 Java 的网络编程框架，能为应用程序管理复杂的网络编程、多线程处理以及并发。Netty 隐藏了样板和底层代码，让业务逻辑保持分离，更加易于复用。使用 Netty 可以得到一个易于使用的 API，让开发人员可以专注于自己的应用程序的独特之处。

本书介绍了 Netty 框架，并展示了如何将它引入到 Java 网络应用程序中。通过对本书的学习，读者能学到如何编写高度可伸缩的应用程序而无需关心底层 API。本书将通过许多动手的例子教读者以异步的方式进行思考，并帮助读者掌握构建大规模网络应用程序的最佳实践。

## 本书主要内容

- 透彻理解 Netty。
- 异步的、事件驱动的编程。
- 使用不同的协议实现服务。
- 覆盖了 Netty 4.x。

本书假设读者熟悉 Java 和基本的网络体系结构。

**Norman Maurer** 是苹果公司的资深软件工程师，同时也是 Netty 的核心开发人员。**Marvin Allen Wolfthal** 是 Dell Services 的顾问，他使用 Netty 实现了多个任务关键型的企业系统。

“第一本关于 Netty 的书……展示了如何构建高性能、低延迟的网络应用程序。”

——摘自 Netty 创始人 Trustin Lee 为本书撰写的序

“高性能的 Java 网络栈——涵盖了从概念到最佳实践的内容。”

——Christian Bach, 网格交易平台

“关于最大限度利用 Netty 的最全面的书。”

——Jürgen Hoffmann, Red Hat

“关于 Netty 框架的极好综述。强烈推荐给所有使用 Java 处理性能敏感的网络 I/O 的人。”

——Yestin Johnson, Impact Radius



人民邮电出版社  
www.epubit.com.cn



异步社区 [www.epubit.com.cn](http://www.epubit.com.cn)  
新浪微博 @人邮异步社区  
投稿/反馈邮箱 [contact@epubit.com.cn](mailto:contact@epubit.com.cn)

美术编辑：董志桢

分类建议：计算机／程序设计／Java  
人民邮电出版社网址：[www.ptpress.com.cn](http://www.ptpress.com.cn)

ISBN 978-7-115-45368-6



9 787115 453686 >