
目录

Introduction	1.1
gPRC 介绍	1.2
资料收集整理	1.2.1
Protocol Buffer 3	1.2.2
gPRC文档	1.3
gRPC官方文档(中文版)	1.3.1
gRPC动机和设计原则	1.3.2
源码导航	1.3.3
基础	1.4
NameResolver	1.4.1
URI术语	1.4.1.1
类NameResolver	1.4.1.2
类DnsNameResolver	1.4.1.3
类DirectAddressNameResolver	1.4.1.4
NameResolver的用法	1.4.1.5
Load Balancer	1.4.2
Metadata	1.4.3
Channel层	1.5
Channel设计与代码实现	1.5.1
类Channel	1.5.1.1
类ManagedChannel	1.5.1.2
类ManagedChannelImpl	1.5.1.3
空闲模式	1.5.1.3.1
InUseStateAggregator	1.5.1.3.2
Name Resolver	1.5.1.3.3
Load Balancer	1.5.1.3.4
Transport	1.5.1.3.5

Executor	1.5.1.3.6
关闭	1.5.1.3.7
Channel Builder设计与代码实现	1.5.2
类ManagedChannelBuilder	1.5.2.1
类AbstractManagedChannelImplBuilder	1.5.2.2
类NettyChannelBuilder	1.5.2.3
Channel Provider设计与代码实现	1.5.3
类ManagedChannelProvider	1.5.3.1
类NettyChannelProvider	1.5.3.2
类CallOptions	1.5.4
Stub层	1.6
类DemoServiceBlockingStub	1.6.1
类AbstractStub	1.6.2
客户端流程	1.7
执行RPC调用	1.7.1
服务器端流程	1.8
状态	1.8.1
类Status	1.8.1.1
状态码详细定义	1.8.1.2
类StatusException	1.8.1.3
异常处理的流程分析	1.8.1.4
实践	1.9
集成Spring Boot	1.9.1
文档生成	1.9.2
支持proto3	1.9.2.1
build插件	1.9.2.2
使用模板定制输出	1.9.2.3
代理	1.9.3
超时	1.9.4
全文标签总览	1.10

gRPC 学习笔记

gRPC 是 google 最新发布(始于2015年2月, 1.0正式版本发布于2016年8月)的开源 RPC 框架, 声称是"一个高性能, 开源, 将移动和HTTP/2放在首位的通用的RPC 框架.". 技术栈非常的新, 基于HTTP/2, netty4.1, proto3, 拥有非常丰富而实用的特性, 堪称新一代RPC框架的典范.

这是个人学习gRPC的笔记, 请点击下面的链接阅读或者下载电子版本:

- 在线阅读
 - [国外服务器](#): gitbook提供的托管, 服务器在国外, 速度比较慢, 偶尔被墙, HTTPS
 - [国内服务器](#): 腾讯云加速, 国内网速极快, 非HTTPS
- [下载pdf格式](#)
- [下载mobi格式](#)
- [下载epub格式](#)

本文内容可以任意转载, 但是需要注明来源并提供链接。

请勿用于商业出版。

google grpc 介绍

=====

grpc是google最新发布(2015年2月底)的开源rpc框架。

按照google的说法，grpc是：

A high performance, open source, general RPC framework that puts mobile and HTTP/2 first.

一个高性能，开源，将移动和HTTP/2放在首位的通用的RPC框架。

特性

HTTP/2

构建于HTTP/2标准，带来很多功能，如：

- bidirectional streaming
- flow control
- header compression
- multiplexing requests over a single TCP connection

mobile

这些特性在移动设备上节约电池使用时间和数据使用，加速服务和运行在云上的web应用。

资料收集整理

网站

- [grpc官网](#)
- [grpc-java](#) gRPC Java 实现.
- [javadoc](#): grpc java 的javadoc地址
- [grpc google groups](#)
- [grpc-ecosystem](#)

文档

- [grpc-common](#) 是官方提供的文档和例子, 但是内容实际是指向下面的[grpc.io](#)上的Documentation.
- [Documentation@grpc.io](#) 是[grpc.io](#)提供的文档, 这个适合入门

Documentation@grpc.io中内容比较重要:

- [overview](#): grpc的overview
- [java教程](#) 是官方提供的针对java的教程

注: 开源中国组织人手翻译这份文档, [gRPC 官方文档中文版](#)

Demo

- [grpc-android-demo](#): android的demo
- [grpc-streaming-demo](#): A quick demo of bi-directional streaming RPC's using grpc, go and python
- [yeyincai/grpc-demo](#): introduces using grpc about encryption 、stream 、oneof 、interceptor 、loadbalance demo <http://blog.csdn.net/yeyincai>

工具

- [grpc-tools](#): Tools useful with gRPC libraries, provided by grpc

项目

- [kafka-pixy](#): gRPC/REST proxy for Kafka
- [grpc-experiments](#): Experiments and proposals for gRPC features.
- [grpc-gateway](#): gRPC to JSON proxy generator
- [LogNet/grpc-spring-boot-starter](#): Spring Boot starter module for gRPC framework.
- [grpc-opentracing](#): OpenTracing is a set of consistent, expressive, vendor-neutral APIs for distributed tracing and context propagation

周边项目

- [grpc-gateway](#) : 是一个基于go语言的项目.

[grpc-gateway](#)是protoc的插件. 它读取gRPC 服务定义, 然后生成一个反向代理服务器, 将RESTful JSON API转为gRPC.

用于帮助为API同时提供gRPC 和 RESTful接口.

这个工具似乎不错,对于某些需要提供restul接口场合可以快速的在grpc接口上转换出来.

- [grpc-docker-library](#): 包含官方gRPC Docker镜像的Git仓库
- [grpc-spring-boot-starter](#): 介绍 [gRPC Spring Boot Starter - SprintBoot](#) 的 gRPC 模块

Google Protocol Buffer

gRPC的一个重要基石就是 Protocol Buffer 3, 这个版本(被称为proto3)是原有 Protocol Buffer 2(被称为proto2)的升级版本, 删除了一部分特性, 优化了对移动设备的支持, 另外增加了对android和ios的支持, 使得gRPC可以顺利的在移动设备上使用。

在中文资料几乎没有的情况下, 决定自己动手翻译。

以下是翻译完成的文档列表：

- 开发指南
- API参考文档

注：proto3的内容在上述文档翻译完成之后, 发现内容比较多, 而且这些内容比较独立。因此提取为单独的一个名为 "[Learning Proto3](#)" 的 [proto3学习笔记](#), 和protocol buffer相关的内容都被转移过去。

gRPC文档

这里收集了一些gRPC的文档，坦白说gRPC目前在文档方面做的非常的不足，尤其在2015年的初期，不过那个时候gRPC才刚开始开源可以理解，而在2016之后文档已经完善了很多。相信后面应该会有更好。

gRPC文档的另外一个严重问题是：几乎找不到中文文档和资料。直到2016年初，开源中国以众包的方式翻译了gRPC的官方文档才终于有所改善。

gRPC官方文档

这是目前唯一的一份中文文档，具体内容见后面的章节。

其他文档

还能找到一些其他的文档，例如：

- [gRPC动机和设计原则](#)

gRPC官方文档(中文版)

注：gRPC的 [官方文档](#)，我自己动手翻译了一部分内容，但是后来看到 [开源中国](#) 组织人手翻译了整份文档，因此我就不再继续，直接转载。

开源中国翻译的文档访问地址：[gRPC 官方文档中文版 v1.0](#)

文档地址

- [gRPC官方文档](#)：英文原文
- [gRPC 官方文档中文版 v1.0](#): 上面文档的中文翻译版本，开源中国提供

读后感

1. 翻译的质量不是太高，当然也凑合能读。
2. 然后gRPC的这份文档，个人感觉，也水了点。做简单入门还凑合，但是也只是能简单入门而已，想更深入一点，这份文档是远远不够的。

gRPC动机和设计原则

注: 官网文档 [gRPC Motivation and Design Principles](#), 我原来自己写了一份简单的读书笔记, 后来发现有同学全文翻译了这篇文章, 就放弃了自己的内容直接转载了。

文档地址

- [gRPC Motivation and Design Principles](#) : 英文原文
- [GRPC的产生动机和设计原则](#): 此文的中文翻译版本

读后感

注: 以下是个人的一点感触

2015年3月的某一天, 第一次接触gRPC, 当时的第一感觉: 这就是我要的东西!!

上面的这个文档中, 在阐述"原则和诉求"时, 有几点是特别打动我的:

服务而非对象、消息而非引用

促进微服务的系统间粗粒度消息交互设计理念, 同时避免分布式对象的陷阱和分布式计算的谬误。

2015年初, 当时正在理解和接受微服务的理念, 发现gRPC在理念上特别符合微服务的想法。后来看gRPC文档时看到上面这句, 才知道原来gRPC的设计理念本来就是冲着微服务去的.....

在这之后的一年中, 我心目中完美的服务化框架慢慢的形成了, 基石就是: 微服务 + gRPC。

普遍并且简单

该基础框架应该在任何流行的开发平台上适用，并且易于被个人在自己的平台上构建。它在CPU和内存有限的设备上也应该切实可行。

"在CPU和内存有限的设备",对我而言就是移动设备了，尤其特指手机和平板。2015年初在唯品会，遇到手机App端和服务端通讯的方案选择问题，当时我特别想把手机App端和服务端这块的通讯机制整合入唯品会的服务化框架，但是因种种原因未能如愿，深以为憾。

当时有一个非常强烈的诉求，就是在如今的移动互联网时代，PC没落手机泛滥，如何可以实施一个服务化框架而无视移动端的需求？

之后就一直在gRPC和Rest之间摇摆，但是我对这两个方案的共同要求，都是可以一套方案打通app到服务器和服务端之间的通讯机制。

2016年，在PPMoney，很有幸，基于gRPC的微服务框架得以开发并实施。

流

存储系统依赖于流和流控来传递大数据集。像语音转文本或股票代码等其它服务，依靠流表达时间相关的消息序列。

虽说目前面对的直接需求中，对流的要求不多，乃至几乎没有。但是我依然看好这个思路，gRPC在框架层次上直接提供对流的支持，想法很好很对路。

元数据交换

常见的横切关注点，如认证或跟踪，依赖数据交换，但这不是服务公共接口中的一部分。部署依赖于他们将这特性以不同速度演进到服务暴露的个别API的能力。

之前都是想办法自己来生成/传输和利用元数据，如今终于gRPC直接提供。还有什么好说，用，用，用！

源码导航

注：内容翻译自 [grpc-java](#) 首页的 [Navigating Around the Source](#)。

从高水平上看，类库有三个不同的层：**Stub/桩**, **Channel/通道** & **Transport/传输**。

Stub

Stub层暴露给大多数开发者，并提供类型安全的绑定到正在适应（adapting）的数据模型/IDL/接口。gRPC带有一个protocol-buffer编译器的 [插件](#) 用来从 `.proto` 文件中生成Stub接口。当然，到其他数据模型/IDL的绑定应该是容易添加并欢迎的。

关键接口

[Stream Observer](#)

Channel

Channel层是传输处理之上的抽象，适合拦截器/装饰器，并比Stub层暴露更多行为给应用。它想让应用框架可以简单的使用这个层来定位横切关注点（address cross-cutting concerns）如日志，监控，认证等。流程控制也在这个层上暴露，容许更多复杂的应用来直接使用它交互。

Common

- [元数据 - headers & trailers](#)
- [状态 - 错误码命名空间 & 处理](#)

Client

- [Channel - 客户端绑定](#)
- [Client Call](#)
- [Client 拦截器](#)

Server

- [Server call handler - analog to Channel on server](#)
- [Server Call](#)

Transport

Transport层承担在线上放置和获取字节的繁重工作。它的接口被抽象到恰好刚刚够容许插入不同的实现。Transport被建模为 `Stream` 工厂。`server stream`和`client stream`之间在接口上的存在差别以整理他们在取消和错误报告上的不同语义。

注意transport层的API被视为gRPC的内部细节，并比在package `io.grpc` 下的core API有更弱的API保证。

gRPC带有三个Transport实现：

1. [基于Netty](#) 的transport是主要的transport实现，基于[Netty](#). 可同时用于客户端和服务端。
2. [基于OkHttp](#) 的transport是轻量级的transport，基于[OkHttp](#). 主要用于Android并只作为客户端。
3. [inProcess](#) transport 是当服务器和客户端在同一个进程内使用使用。用于测试。

Common

- [Stream](#)
- [Stream Listener](#)

Client

- [Client Stream](#)
- [Client Stream Listener](#)

Server

- [Server Stream](#)
- [Server Stream Listener](#)

基础

NameResolver

NameResolver 是可拔插的组件，用于解析目标 URI 并返回地址给调用者。

NameResolver 使用 URI 的 `scheme` 来检测是否可以解析它，再使用 `scheme` 后面的组件来做实际处理。

目标的地址和属性可能随着时间的过去发生修改，因此调用者注册 `Listener` 来接收持续更新。

URI 术语

注：后面大量章节涉及到 **URI** 标准中的诸多术语，为了概念清晰，单独开一章来介绍。

URI 的语法

通用 URI 和绝对 URI 的语法参考最初定义在 [RFC 2396](#), 发布于1998年，并定稿于 RFC 3986，发布于 2005年。

通用 URI 的形式如下：

```
scheme:[//[user:password@]host[:port]][/]path[?query][#fragment]
```

它包括：

- **scheme**

由一系列字符组成，以字母开头，跟着有字母，数字，加号(+)，点号(.)或者中划线(-)的任何组合。

虽然 **scheme** 是大小写不敏感，但是权威形式是小写并注明特定 **scheme** 必须也同样是小写。后面跟一个冒号(:)。流行的 **scheme** 例子包括 `http`，`ftp`，`mailto`，`file`，和 `data`。URI **scheme** 应该在 [Internet Assigned Numbers Authority \(IANA\)](#) 注册, 虽然不注册的 **scheme** 实际也在使用。

- 双斜杠(//)

某些 **scheme** 要求有这个，而有些 **scheme** 不要求。当 **authority** 部分缺失时，**path** 部分不能以双斜杠开始。

- **authority** 部分

包括：

- 可选的认证部分，有用户名和密码，冒号分隔，带一个@符号
- **host**, 包括一个被注册名字(包括但是不限于hostname)，或者 IP 地址。
IPv4 地址必须以点号分割的形式，而 IPv6 地址必须加[]符号。

- 可选的端口号，和hostname之间用冒号分隔。

- 路径

包含数据，通常以层次形式组织，表现为一系列斜杠分隔的部分。这样一个序列可能类似或者映射到文件系统路径，但是并不暗示一定和某个路径有关系。如果有 **authority** 部分，则路径必须以单斜杠开始，

- 可选的**query**

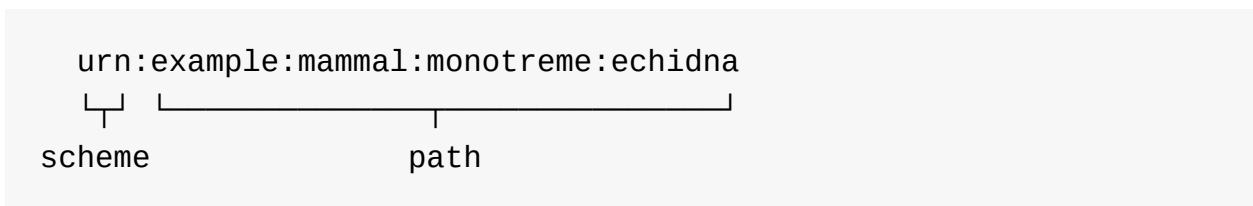
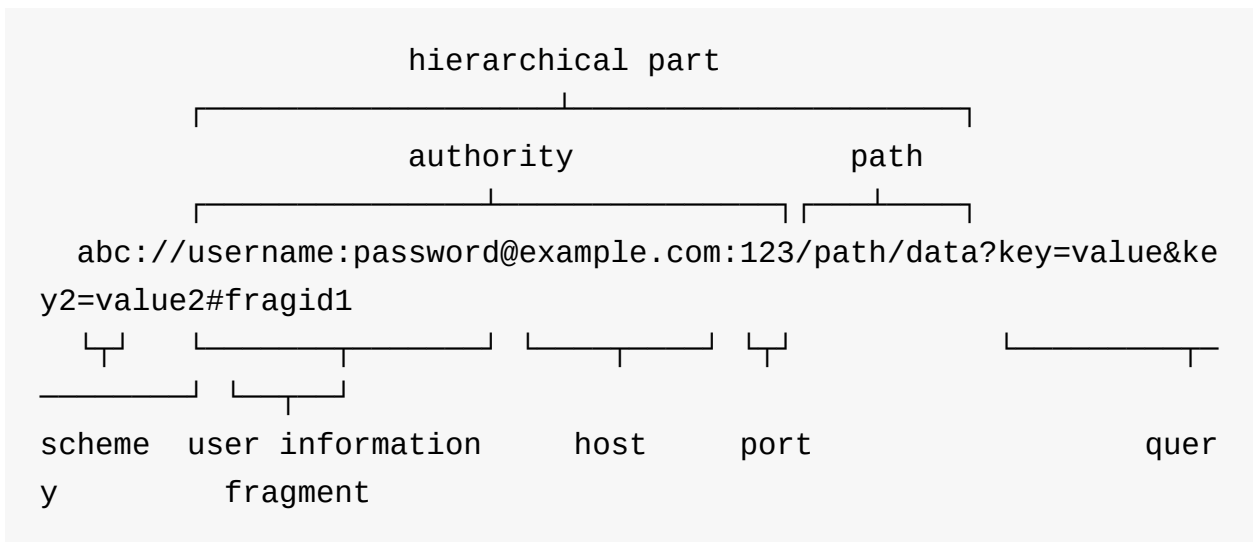
和之前的部分以问号(?)分隔，包含非层次数据的 **query string**。它的语法没有很好的定义，但是习惯上通常是一系列的属性值对，以分隔符分隔。

- 可选的**fragment/片段***

和之前的部分以#分隔。**fragment** 包括一个 **fragment identifier / 片段标识**，提供到间接资源的引导，就像文章中的章节头，通过URI的剩余部分定位。当首要资源是 HTML 文档时，**fragment** 通常是一个特别原始的 id 属性，而web浏览器将滚动这个元素到视野中。

例子

下面的图片展示了两种URI的例子，和他们的组成部分。



特别强调

authority代表URI中的 `[userinfo@]host[:port]`，包括host(或者ip)和可选的port和userinfo。

这个术语平时用的不多，但是在gRPC中频繁出现，请牢记：**authority** 代表 `username:password@example.com:123` 这一段。

参考资料

- https://en.wikipedia.org/wiki/Uniform_Resource_Identifier

类 NameResolver

类定义

NameResolver 定义为一个 **abstract** 类，而不是 **interface**：

```
public abstract class NameResolver {}
```

类方法

getServiceAuthority()

返回用于验证与服务器的连接的权限(authority)。必须来自受信任的来源，因为如果权限被篡改，RPC可能被发送到攻击者，泄露敏感用户数据。

实现必须以不阻塞的方式生成它，通常在一行中(in line)，必须保持不变。使用同样的参数从同一个的 **factory** 中创建出来的 NameResolver 必须返回相同的 **authority**。

```
public abstract String getServiceAuthority();
```

start()

开始解析。listener 用于接收目标的更新。

```
public abstract void start(Listener listener);
```

shutdown()

停止解析。listener 的更新将会停止。

```
public abstract void shutdown();
```

refresh()

重新解析名字。

只能在 `start()` 方法被调用之后调用。

这里仅仅是一个暗示。实现类将它作为一个信号，但是可能不会立即开始解析。它绝不抛出异常。

默认实现是什么都不做。

```
public void refresh() {}
```

内部类 Factory

创建 NameResolver 实例的工厂类。

```
public abstract static class Factory {  
    /**  
     * 端口号，用于当目标或者底层命名系统没有提供端口号的情况  
     */  
    public static final Attributes.Key<Integer> PARAMS_DEFAULT_P  
ORT =  
        Attributes.Key.of("params-default-port");  
}
```

newNameResolver()

创建 NameResolver 用于给定的目标URI，或者在给定URI无法被这个 factory 解析时返回 null。决定应该仅仅基于 URI 的 scheme。

参数 `targetUri` 表示要解析的目标 URI，而这个 URI 的 scheme 必须不能为 null。

参数 `params` 是可选参数。权威key在 Factory 中以 `PARAMS_*` 字段定义。

```
public abstract NameResolver newNameResolver(Uri targetUri, Attr  
ibutes params);
```

getDefaultScheme()

返回默认 scheme，当 ManagedChannelBuilder.forTarget(String) 方法被给到 authority 字符串而不是符合的 URI 时，用于构建 URI。

```
public abstract String getDefaultScheme();
```

内部接口 Listener

接收地址更新。

所有的方法都要求快速返回。

```
public interface Listener {}
```

onUpdate()

注意：已经被废弃，改用 onAddresses() 方法。

```
@Deprecated
void onUpdate(List<ResolvedServerInfoGroup> servers, Attributes
attributes);
```

onAddresses()

处理被解析的地址和配置的更新。

实现不可以修改给定的参数 servers。

参数 servers 指被解析的服务器地址。空列表将触发 onError() 方法。

```
void onAddresses(List<EquivalentAddressGroup> servers, Attribute
s attributes);
```

onError()

处理从 **resolver** 而来的错误。

参数 **error** 为非正常的状态

```
void onError(Status error);
```


类 DnsNameResolver

类定义

DnsNameResolver 是 `io.grpc.internal` 下的类，包级私有，通过 `DnsNameResolverFactory` 类来创建。

研究一下它的实现，以便理解 `NameResolver` 的使用。

```
package io.grpc.internal;  
  
class DnsNameResolver extends NameResolver {}
```

属性和构造函数

属性比较多，先只看和 URI 相关的几个属性 `authority/host/port`：

```

private final String authority;
private final String host;
private final int port;

DnsNameResolver(@Nullable String nsAuthority, String name, Attributes params,
    Resource<ScheduledExecutorService> timerServiceResource,
    Resource<ExecutorService> executorResource,
    ProxyDetector proxyDetector) {

    // 必须在 name 前加上"//"，否则将被当成含糊的URI，导致生成的URI的authority和host会变成null。
    URI nameUri = URI.create("//" + name);
    // 从生成的URI中得到 authority 并赋值，如果为空则报错
    authority = Preconditions.checkNotNull(nameUri.getAuthority()),
        "nameUri (%s) doesn't have an authority", nameUri);
    // 同样方式得到host，如果为空则报错
    host = Preconditions.checkNotNull(nameUri.getHost(), "host");
;
    // 端口特殊一点，因为可以通过params参数传入默认端口
    if (nameUri.getPort() == -1) {
        // 如果name中没有包含端口，则尝试从参数中获取默认端口
        Integer defaultPort = params.get(NameResolver.Factory.PARAMS_DEFAULT_PORT);
        if (defaultPort != null) {
            port = defaultPort;
        } else {
            throw new IllegalArgumentException(
                "name '" + name + "' doesn't contain a port, and default port is not set in params");
        }
    } else {
        port = nameUri.getPort();
    }
}

```

构造函数中还传入了两个和Executor相关的参数，构造函数只是简单的保存起来：

```
private final Resource<ScheduledExecutorService> timerServiceResource;
private final Resource<ExecutorService> executorResource;

DnsNameResolver(@Nullable String nsAuthority, String name, Attributes params,
    .....
    this.timerServiceResource = timerServiceResource;
    this.executorResource = executorResource;
    .....
}
```

方法实现

NameResolver 定义的方法

getServiceAuthority()

按照要求，这个方法直接返回在构造函数中就设置好的 `authority` 属性。考虑到 `authority` 属性是 `final` 的，因此也满足 `NameResolver` 接口中要求的：“实现必须非阻塞式的生成它，而且必须保持不变。使用同样的参数从同一个的 `factory` 中创建出来的 `NameResolver` 必须返回相同的 `authority`”。

```
public final String getServiceAuthority() {
    return authority;
}
```

start()

按照要求，`start()` 开始解析。`listener` 用于接收目标的更新。

实现中，`this.listener` 属性用于保存传入的 `listener`，而在 `start()` 时，`timerService` 和 `executor` 两个属性才会从之前传入的 `timerServiceResource/executorResource` 这两个 `SharedResourceHolder` 中获取实际的对象实例。

```
public final synchronized void start(Listener listener) {  
    // 先检查之前没有start过，判断依据是 this.listener 是否已有赋值  
    Preconditions.checkState(this.listener == null, "already started");  
    timerService = SharedResourceHolder.get(timerServiceResource);  
    executor = SharedResourceHolder.get(executorResource);  
    // 为 this.listener 赋值，配合上面的检查  
    this.listener = Preconditions.checkNotNull(listener, "listener");  
    resolve();  
}
```

resolve()方法开始做实际的解析，具体内容后面再看。

refresh()

DnsNameResolver 的 refresh() 实现直接调用了 resolve() 方法，和 start() 方法相比，start()中只是多了开始时的检查和resource获取，后面都是同样的调用 resolve() 方法。

```
public final synchronized void refresh() {  
    // refresh()方法必须在 start() 之后调用，因此这里做了检查，同样判断依据是 listener 属性  
    Preconditions.checkState(listener != null, "not started");  
    resolve();  
}
```

shutdown()

按照要求，shutdown()方法将停止解析，同时更新 listener 将会停止。实现中，将 shutdown属性用于标记是否已经关闭。

```
@GuardedBy("this")
private boolean shutdown;

public final synchronized void shutdown() {
    // 通过 shutdown 属性来判断是否已经关闭
    if (shutdown) {
        return;
    }
    shutdown = true;
    if (resolutionTask != null) {
        // 如果 resolutionTask 不为空，取消它
        resolutionTask.cancel(false);
    }
    if (timerService != null) {
        // 释放 timerService 资源，返回null将清理属性 timerService
        timerService = SharedResourceHolder.release(timerServiceResource, timerService);
    }
    if (executor != null) {
        // 释放 executor 资源，返回null将清理属性 executor
        executor = SharedResourceHolder.release(executorResource, executor);
    }
}
```

解析的实际实现

从 `resolve()` 方法开始：

```
private void resolve() {  
    // resolving 属性和 shutdown 协助判断一下状态  
    if (resolving || shutdown) {  
        return;  
    }  
    // 将 resolutionRunnable 作为任务扔给executor  
    // 然后方法返回，异步做解析  
    // 这样 start()和 refresh() 方法就都是快速返回，异步解析后通过 listener 做数据更新  
    executor.execute(resolutionRunnable);  
}
```

继续看 `resolutionRunnable` 的实现，代码有点长，先排除各种细节处理，只看主流程：

```

private final Runnable resolutionRunnable = new Runnable() {
    @Override
    public void run() {
        //忽略此处的状态检查代码和grpc proxy处理代码
        .....
        ResolutionResults resolvedInetAddrs;
        try {
            // step1. 将host解析为ResolutionResults
            // 新版本引入了一个 delegateResolver，细节稍后再看
            // 开始对 host 做 dns 解析，得到解析结果
            resolvedInetAddrs = delegateResolver.resolve(host);
        } catch (Exception e) {
            ..... // 异常处理的流程后面再细看
            return;
        }
        // 解析出来的每个地址组成一个 EAG
        ArrayList<EquivalentAddressGroup> servers = new ArrayList<EquivalentAddressGroup>();
        for (InetAddress inetAddr : resolvedInetAddrs.addresses)
        {
            step2. 将每个 InetAddress 格式的IP地址包装为 EquivalentAddressGroup 对象
            servers.add(new EquivalentAddressGroup(new InetSocketAddress(inetAddr, port)));
        }
        // 跳过balancerAddresses和TXT的处理
        // step3. 通知listener，有数据更新
        savedListener.onAddresses(servers, attrs.build());
    }
}

```

主流程就是上面注释中的3个步骤：

1. 解析地址
2. 包装格式
3. 通知listener

注意这个工作是在异步线程中进行的，无法直接return结果，只能通过listener。

再继续看错误流程，如果解析地址失败：

```
try {
    resolvedInetAddrs = delegateResolver.resolve(host);
} catch (Exception e) {
    // 遇到无法解析的情况
    synchronized (DnsNameResolver.this) {
        if (shutdown) {
            // 如果此时已经要求 shutdown，则不用继续处理，直接return
            return;
        }
        // 因为在生产中 timerService 是一个单线程的 GrpcUtil.TIMER_SERVICE
        // 我们需要将这个阻塞的工作交给 executor
        resolutionTask =
            timerService.schedule(new LogExceptionRunnable(resolutionRunnableOnExecutor), 1, TimeUnit.MINUTES);
    }
    // 通知 listener 遇到错误
    savedListener.onError(Status.UNAVAILABLE.withDescription(
        "Unable to resolve host " + host).withCause(e));
    return;
}
```

上面的代码，在处理解析失败时，做了两件事情：

1. 通知 listener 出错了
2. 安排了一个 resolutionTask

出错了通知 listener 这个容易理解，resolutionTask 是做什么呢？我们细看 resolutionTask 相关的代码：

```
// resolutionTask 的定义，一个标准的 ScheduledFuture
private ScheduledFuture< ? > resolutionTask;
.....

// 唯一一个赋值的地方，就是当解析出错时，也就是上面的处理
resolutionTask = timerService.schedule(new LogExceptionRunnable(
    resolutionRunnableOnExecutor), 1, TimeUnit.MINUTES);
```


给 timerService 提交一个 LogExceptionRunnable 的任务，要求延迟1分钟执行，然后将得到的 feature 保存为 resolutionTask。这个 resolutionTask 在 shutdown() 方法和 resolutionRunnable 的 run()方法中有细节处理。

先看看 LogExceptionRunnable 的实现：

```
// 对 Runnable 的简单包裹，用于记录它抛出的任何异常，在重新抛出之前
public final class LogExceptionRunnable implements Runnable {
    // 构造函数只是简单的保存传入的task
    public LogExceptionRunnable(Runnable task) {
        this.task = checkNotNull(task);
    }
    public void run() {
        try {
            // 执行task
            task.run();
        } catch (Throwable t) {
            // 捕获异常，先打印日志，这个是主要目的了
            log.log(Level.SEVERE, "Exception while executing runnable "
+ task, t);
            // 如果是RuntimeException或者Error，就直接原样抛出去
            MoreThrowables.throwIfUnchecked(t);
            // 否则就生成一个新的AssertionError抛出去
            throw new AssertionError(t);
        }
    }
}
```

只是简单的执行 task 并在出错时记录日志，而这里的 task 是 resolutionRunnableOnExecutor，所以关键还是看 resolutionRunnableOnExecutor 里面的实现内容：

```
private final Runnable resolutionRunnableOnExecutor = new Runnable() {
    @Override
    public void run() {
        synchronized (DnsNameResolver.this) {
            if (!shutdown) {
                // 再执行一次 resolutionRunnable，这里和 resolve() 函数差不多
                的功能
                executor.execute(resolutionRunnable);
            }
        }
    }
};
```

现在错误流程就清晰了：

1. 在解析失败时，就会给 timerService 安排一个一分钟之后执行的任务 resolutionTask
2. 在 resolutionTask 中，将重新用 executor 跑一次 resolutionRunnable
3. 如果继续解析失败，则循环上述过程

即解析失败则每隔一分钟尝试一次，直到成功。

注意在 resolutionRunnable 中，每次发现 resolutionTask 存在就会先 cancel 掉它，然后置为null：

```
if (resolutionTask != null) {
    resolutionTask.cancel(false);
    resolutionTask = null;
}
```

所以上述的循环，只是在每次解析失败时，一旦解析成功，就会跳出循环。

类 DirectAddressNameResolver

在类 AbstractManagedChannelImplBuilder 中有一个内部类，
DirectAddressNameResolverFactory，这里实现了一个 NameResolver，用于处理
DirectAddress。

Factory 的代码

```
private static final String DIRECT_ADDRESS_SCHEME = "directaddress";

private static class DirectAddressNameResolverFactory extends NameResolver.Factory {
    final SocketAddress address;
    final String authority;

    // 构造函数中直接提供 address 和 authority
    DirectAddressNameResolverFactory(SocketAddress address, String authority) {
        this.address = address;
        this.authority = authority;
    }

    @Override
    public NameResolver newNameResolver(URI notUsedUri, Attributes params) {
        return new NameResolver() {
            .....// 细节后面看
        }
    }

    @Override
    public String getDefaultScheme() {
        // 默认的scheme是固定的 "directaddress"
        return DIRECT_ADDRESS_SCHEME;
    }
}
```

NameResolver的实现

DirectAddressNameResolverFactory的NameResolver的实现是一个内部匿名类：

```
public NameResolver newNameResolver(Uri notUsedUri, Attributes parameters) {
    return new NameResolver() {
        @Override
        public String getServiceAuthority() {
            // 直接返回factory中传入并保存的authority
            return authority;
        }

        @Override
        public void start(final Listener listener) {
            // 不用解析，直接将 factory 中传入并保存的 address 给出去
            listener.onAddresses(
                Collections.singletonList(new EquivalentAddressGroup(address)),
                Attributes.EMPTY);
        }

        @Override
        public void shutdown() {}
    };
}
```

总结

这个实现够简单，不过考虑到平时也是有需要用到直接地址的。

NameResolver的用法

使用场合

类ManagedChannelBuilder

类ManagedChannelBuilder 的 nameResolverFactory() 方法用于在构建 channel 时指定需要的 resolverFactory :

```
@ExperimentalApi
public abstract T nameResolverFactory(NameResolver.Factory resolverFactory);
```

为channel提供一个定制的 NameResolver.Factory。如果这个方法没有被调用，builder将在全局解析器注册(global resolver registry)中为提供的目标寻找一个工厂。

工作方式

NameResolver的工作方式，和 Channel / Transport / Load Balancer 有密切关系，详细细节请见后面的章节：

- [name resolver @ ManagedChannelImpl](#)

Load Balancer

类Metadata

提供对读取和写入元数据数值的访问，元数据数值在调用期间交换。

key 容许关联到多个值。

这个类不是线程安全，实现应该保证 **header** 的读取和写入不在多个线程中并发发生。

类定义

```
package io.grpc;

@NotThreadSafe
public final class Metadata{}
```

类属性

```
// 所有二进制 header 在他们的名字中应该有这个后缀。相反也是。
// 它的值是"-bin"。ASCII header 的名字一定不能以这个结尾。
public static final String BINARY_HEADER_SUFFIX = "-bin";

// 保存的数据
private byte[][] namesAndValues;
// 当前未缩放的 header 数量
private int size;
```

构造函数


```
// 构造函数，当应用层想发送元数据时调用
public Metadata() {}

// 构造函数，当传输层接收到二进制元数据时调用。
Metadata(byte[]... binaryValues) {
    // 注意这里长度除以 2 了
    this(binaryValues.length / 2, binaryValues);
}
Metadata(int usedNames, byte[]... binaryValues) {
    size = usedNames;
    namesAndValues = binaryValues;
}
```

和size相关的几个私有方法：

1. len()

```
private int len() {
    return size * 2;
}
```

2. headerCount(): 返回在这个元数据中的键值 header 的总数量，包含重复

```
public int headerCount() {
    return size;
}
```

存取数据的方法

1. containsKey(): 如果给定的key有值则返回true，注意这里没有检查value是否可能为空列表

```
public boolean containsKey(Key<?> key) {
    for (int i = 0; i < size; i++) {
        if (bytesEqual(key.asciiName(), name(i))) {
            return true;
        }
    }
    return false;
}
```

注意这里是线性搜索，因此如果后面跟有 `get()` 或者 `getAll()` 方法，最好直接调用他们然后检查返回值是否为 `null`。

2. `get()`: 返回最后一个用给定 `name` 添加的元数据项，作为 `T` 解析，如果没有则返回 `null`

```
public < T > T get(Key < T > key) {
    for (int i = size - 1; i >= 0; i--) {
        if (bytesEqual(key.asciiName(), name(i))) {
            return key.parseBytes(value(i));
        }
    }
    return null;
}
```

3. `getAll()`: 返回所有用给定 `name` 添加的元数据项，作为 `T` 解析，如果没有则返回 `null`。顺序和接收到的一致。

```
public < T> Iterable< T> getAll(final Key< T> key) {
    for (int i = 0; i < size; i++) {
        if (bytesEqual(key.asciiName(), name(i))) {
            return new IterableAt<T>(key, i);
        }
    }
    return null;
}
```

这里一旦返现有 **key** 匹配，就直接new 一个 **IterableAt** 对象。后面对所有 同样 **name** 的游离是通过 **IterableAt** 来实现。

4. **keys()**: 返回存储中的所有**key**的不可变集合

```
@SuppressWarnings("deprecation") // The String ctor is deprecated, but fast.
public Set<String> keys() {
    if (isEmpty()) {
        return Collections.emptySet();
    }
    Set<String> ks = new HashSet<String>(size);
    for (int i = 0; i < size; i++) {
        ks.add(new String(name(i), 0 /* hbyte */));
    }
    // immutable in case we decide to change the implementation later.
    return Collections.unmodifiableSet(ks);
}
```

实现逻辑很简单，但是有意思的是对 **String** 构造方法的使用。

5. **put()**: 添加键值对。如果 **key** 已经有值，值被添加到最后。同一个**key**的重复的值是容许的。

```
public < T > void put(Key< T > key, T value) {
    Preconditions.checkNotNull(key, "key");
    Preconditions.checkNotNull(value, "value");
    maybeExpand();
    name(size, key.asciiName());
    value(size, key.toBytes(value));
    size++;
}
```

6. **remove()**: 删除**key**的第一个出现的**value**，如果删除成功返回**true**，如果没有找到返回**false**

```
public < T> boolean remove(Key< T> key, T value) {
    Preconditions.checkNotNull(key, "key");
    Preconditions.checkNotNull(value, "value");
    for (int i = 0; i < size; i++) {
        // 从头开始游历
        if (!bytesEqual(key.asciiName(), name(i))) {
            // 找 匹配的name
            continue;
        }
        @SuppressWarnings("unchecked")
        T stored = key.parseBytes(value(i));
        if (!value.equals(stored)) {
            // 如果 value 不匹配，跳过
            continue;
        }

        // name 和 value 都匹配了，准备做删除
        int writeIdx = i * 2;
        int readIdx = (i + 1) * 2;
        int readLen = len() - readIdx;
        // 将后面的数据向前复制
        System.arraycopy(namesAndValues, readIdx, namesAndV
alues, writeIdx, readLen);
        size -= 1;
        // 将最后一个位置的数据设置为null
        name(size, null);
        value(size, null);
        return true;
    }
    return false;
}
```

7. `removeAll()`: 删除给定key的所有值并返回这些被删除的值。如果没有找到值，则返回null

```

public <T> Iterable<T> removeAll(final Key<T> key) {
    if (isEmpty()) {
        return null;
    }
    int writeIdx = 0;
    int readIdx = 0;
    List<T> ret = null;
    for (; readIdx < size; readIdx++) {
        if (bytesEqual(key.asciiName(), name(readIdx))) {
            ret = ret != null ? ret : new LinkedList<T>();
            ret.add(key.parseBytes(value(readIdx)));
            continue;
        }
        name(writeIdx, name(readIdx));
        value(writeIdx, value(readIdx));
        writeIdx++;
    }
    int newSize = writeIdx;
    // Multiply by two since namesAndValues is interleaved.
    Arrays.fill(namesAndValues, writeIdx * 2, len(), null);
    size = newSize;
    return ret;
}

```

8. `discardAll()`: 删除给定key的所有值但是不返回这些被删除的值。相比 `removeAll()` 方法有细微的性能提升，如果不需要使用之前的值。

内部定义的类

Marshaller

定义了两个 `Marshaller` 的 `interface`，分别用于处理二进制和ASCII字符的值：

1. `BinaryMarshaller`：用于序列化为原始二进制的元数据值的装配器
2. `AsciiMarshaller`: 用于序列化为 ASCII 字符的元数据值的装配器，值只包含下列字符：

- 空格：0x20，但是不能在值的开头或者末尾。开始或者末尾的空白字符可

能被去除。

- ASCII 可见字符 (0x21-0x7E)

然后实现了这两个接口的内部匿名类：

```
public static final BinaryMarshaller<byte[]> BINARY_BYTE_MARSHALLER = new BinaryMarshaller<byte[]>() {};  
  
public static final AsciiMarshaller<String> ASCII_STRING_MARSHALLER = new AsciiMarshaller<String>() {};  
  
static final AsciiMarshaller<Integer> INTEGER_MARSHALLER = new AsciiMarshaller<Integer>() {};
```

内部类 **Metadata.Key**

元数据项的key。考虑到元数据的解析和序列化。

key名称中的有效字符

在key的名字中仅容许下列ASCII字符：

- 数字：0-9
- 大写字符：A-Z（标准化到小写）
- 小写字符：a-z
- 特殊字符：-_.

这是 HTTP 字段名规则的一个严格子集。应用不应该发送或者接收带有非法key名称的元数据。当然，gRPC 类库可能加工任何接收到的元数据，即使它不遵守上面的限制。此外，如果元数据包含不遵守的字段名，他们也将被发送。在这种方式下，未知元数据字段被解析，序列化和加工，但是从不中断。他们类似protobuf的未知字段。

注意，有一个有效的 HTTP/2 token字符的子集，被定义在 RFC7230 3.2.6 节和 RFC5234 B.1 节。

- [Wire Spec](#)
- [RFC7230](#)

- [RFC5234](#)

```
// 字段名的有效字符被定义在 RFC7230 和 RFC5234
private static final BitSet VALID_T_CHARS = generateValidTChars(
);

//在这里看到，仅有三个特殊字符，数字和小写字母是有效的
private static BitSet generateValidTChars() {
    BitSet valid = new BitSet(0x7f);
    valid.set('-');
    valid.set('_');
    valid.set('.');
    for (char c = '0'; c <= '9'; c++) {
        valid.set(c);
    }
    // 仅在标准化之后验证，因此排除大写字母
    for (char c = 'a'; c <= 'z'; c++) {
        valid.set(c);
    }
    return valid;
}
```

`validateName()`方法用于验证输入的字符串(会先转为小写)是否有效：

```
private static String validateName(String n) {
    checkNotNull(n);
    checkArgument(n.length() != 0, "token must have at least 1 tchar");
    for (int i = 0; i < n.length(); i++) {
        char tChar = n.charAt(i);
        // TODO(notcarl): remove this hack once pseudo headers are properly handled
        if (tChar == ':' && i == 0) {
            continue;
        }

        checkArgument(VALID_T_CHARS.get(tChar),
            "Invalid character '%s' in key name '%s'", tChar, n);
    }
    return n;
}
```

Key的类定义

```
public abstract static class Key<T> {}
```

这是一个abstract类，后面将看到它的几个子类实现。

Key的属性和构造函数


```
private final String originalName;
private final String name;
private final byte[] nameBytes;

private Key(String name) {
    // 原始名字是构造函数中传递过来的原始输入
    this.originalName = checkNotNull(name);
    // Intern the result for faster string identity checking.
    // 有效名字是经过处理并验证有效的名字：这里的处理就是简单的改为小写
    this.name = validateName(this.originalName.toLowerCase(Locale.
ROOT)).intern();
    // nameBytes是有效名字的字节数组
    this.nameBytes = this.name.getBytes(US_ASCII);
}
```

Key的抽象方法

定义了两个方法用来做元数据和字节数组之间的转换和解析：

```
// 将元数据序列化到byte数组
abstract byte[] toBytes(T value);

// 从byte数组解析被序列化的元数据值
abstract T parseBytes(byte[] serialized);
```

Key的子类实现

1. BinaryKey

```
private static class BinaryKey<T> extends Key<T> {
    private final BinaryMarshaller<T> marshaller;

    // 构造函数传入一个BinaryMarshaller
    private BinaryKey(String name, BinaryMarshaller<T> marshaller) {
        .....
        this.marshaller = checkNotNull(marshaller, "marshaller is null");
    }

    @Override
    byte[] toBytes(T value) {
        // 使用这个BinaryMarshaller来转换 Key
        return marshaller.toBytes(value);
    }

    @Override
    T parseBytes(byte[] serialized) {
        // 使用这个BinaryMarshaller来解析 Key
        return marshaller.parseBytes(serialized);
    }
}
```

2. AsciiKey：类似，这次使用的是 AsciiMarshaller

Channel层

Channel 到概念上的端点的虚拟连接，用于执行RPC。通道可以根据配置，负载等自由地实现与端点零或多个实际连接。通道也可以自由地确定要使用的实际端点，并且可以在每次RPC上进行更改，从而允许客户端负载平衡。应用程序通常期望使用存根(stub)，而不是直接调用这个类。

应用可以通过使用 **ClientInterceptor** 装饰 **Channel** 实现来为 **stub** 添加常见的切面行为。预计大多数应用代码不会直接使用此类，而是使用已绑定到在应用初始化期间装饰好的 **Channel** 上的存根。

注：上面两段内容翻译自类 `io.grpc.Channel` 的 javadoc.

Channel设计与代码实现

类Channel

类定义

Channel 是一个 abstract class，位于package "io.grpc"，在 grpc-core 这个jar包中。

```
package io.grpc;

import javax.annotation.concurrent.ThreadSafe;

@ThreadSafe
public abstract class Channel {
    .....
}
```

通过 @ThreadSafe 注解标明 Channel 是线程安全的。

方法定义

ClientCall() 方法

```
public abstract <RequestT, ResponseT> ClientCall<RequestT, ResponseT> newCall(
    MethodDescriptor<RequestT, ResponseT> methodDescriptor, CallOptions callOptions);
```

构建一个用于远程操作的 ClientCall 对象，通过给定的 MethodDescriptor 来指定。返回的 ClientCall 对象不会触发任何远程行为，直到 ClientCall.start(ClientCall.Listener, Metadata) 方法被调用。

authority() 方法

```
public abstract String authority();
```

这个 Channel 连接到的目的地的 authority。通常是以 "host:port" 格式。

类ManagedChannel

类ManagedChannel 在 Channel 的基础上提供生命周期管理的功能。

实际实现式就是添加了 shutdown()/shutdownNow() 方法用于关闭 Channel，isShutdown()/isTerminated() 方法用于检测 Channel 状态，以及 awaitTermination() 方法用于等待关闭操作完成。

类定义

```
package io.grpc;  
  
public abstract class ManagedChannel extends Channel {}
```

依然是 abstract class，还继续在 package io.grpc 中。

类方法

shutdown() 方法

```
public abstract ManagedChannel shutdown();
```

发起一个有组织的关闭，期间已经存在的调用将继续，而新的调用将被立即取消。

shutdownNow() 方法

```
public abstract ManagedChannel shutdownNow();
```

发起一个强制的关闭，期间已经存在的调用和新的调用都将被取消。虽然是强制，关闭过程依然不是即可生效;如果在这个方法返回后立即调用 isTerminated() 方法，将可能返回 false。

isShutdown() 方法

```
public abstract boolean isShutdown();
```

返回channel是否是关闭。关闭的channel立即取消任何新的调用，但是将继续有一些正在处理的调用。

isTerminated() 方法

```
public abstract boolean isTerminated();
```

返回channel是否是结束。结束的 channel 没有运行中的调用，并且相关的资源已经释放（像TCP连接）。

awaitTermination() 方法

```
public abstract boolean awaitTermination(long timeout, TimeUnit  
unit) throws InterruptedException;
```

等待 channel 变成结束，如果超时时间达到则放弃。

返回值表明 channel 是否结束，和 isTerminated() 方法一致。

类 **ManagedChannelImpl**

空闲模式

在最新的版本(1.0.0-pre1)中，gRPC 引入了 Channel 的空闲模式(idle mode)。

工作方式(初步)

在 InUseStateAggregator 中，控制 Channel (准备)进入空闲模式和退出空闲模式：

```
final InUseStateAggregator<Object> inUseStateAggregator = new InUseStateAggregator<Object>() {

    @Override
    void handleInUse() {
        // 被使用时，就退出空闲模式
        exitIdleMode();
    }

    @Override
    void handleNotInUse() {
        if (shutdown) {
            return;
        }
        // 如果不被使用，就开始安排定时器准备进入空闲模式
        rescheduleIdleTimer();
    }
};
```

idle 相关属性

```
static final long IDLE_TIMEOUT_MILLIS_DISABLE = -1;

/** 进入空闲模式的超时时间 */
private final long idleTimeoutMillis;
```

idle timer

```
// 不能为null，一定会被 channelExecutor 使用
@Nullable
private ScheduledFuture< ?> idleModeTimerFuture;

// 不能为null，一定会被 channelExecutor 使用
@Nullable
private IdleModeTimer idleModeTimer;
```

类 IdleModeTimer 实际是一个 Runnable (按说取名应该是IdleModeTimerTask才对)，用于使 Channel 进入空闲模式，并关闭以下内容：

1. Name Resolver
2. Load Balancer
3. subchannelPicker

```

// 由 channelExecutor 运行
private class IdleModeTimer implements Runnable {
    // 仅仅由 channelExecutor 修改
    boolean cancelled;

    @Override
    public void run() {
        if (cancelled) {
            // 检测到竞争： 这个任务在 channelExecutor 中安排在 cancelIdleTimer() 之前
            // 需要取消timer
            return;
        }
        log.log(Level.FINE, "[{0}] Entering idle mode", getLogId());
        // nameResolver 和 loadBalancer 保证不为null。如果他们中的任何一个
        // 是 null，
        // 要不是 idleModeTimer 在没有退出空闲模式的情况下运行了两次，
        // 就是 shutdown() 中的任务没有取消 idleModeTimer，两者都是bug

        // 1. 关闭当前的 nameResolver，并重新创建一个新的 nameResolver

        nameResolver.shutdown();
        nameResolver = getNameResolver(target, nameResolverFactory, nameResolverParams);
        // 2. 关闭当前的 loadBalancer 并置为 null
        loadBalancer.shutdown();
        loadBalancer = null;
        // 3. 当前 subchannelPicker 置为 null
        subchannelPicker = null;
    }
}

```

注：没有看懂，为什么要调用 `getNameResolver()` 方法来创建新的 `nameResolver`？

退出空闲模式

让 Channel 退出空闲模式，如果它处于空闲模式中。返回一个新的可以用于处理新请求的 LoadBalancer。如果 Channel 被关闭则返回null。

```
//让 channel 退出空闲模式，如果它处于空闲模式
//必须由 channelExecutor 调用
void exitIdleMode() {
    if (shutdown.get()) {
        return;
    }
    if (inUseStateAggregator.isInUse()) {
        // 立即取消 timer，这样由于 timer 导致的竞争不会将 channel 设置
        // 为空闲
        // 注： 如果正在使用中，则后台会有一个idle timer，需要取消这个ti
        // mer
        cancelIdleTimer();
    } else {
        // exitIdleMode() 可能在 inUseStateAggregator.handleNotIn
        // Use() 之外被调用，此时isInUse() == false
        // 在这种情况下我们依然需要安排 timer
        rescheduleIdleTimer();
    }
    if (loadBalancer != null) {
        return;
    }
    log.log(Level.FINE, "[{0}] Exiting idle mode", getLogId());
    // 1. 创建新的 loadBalancer
    LbHelperImpl helper = new LbHelperImpl(nameResolver);
    helper.lb = loadBalancerFactory.newLoadBalancer(helper);
    this.loadBalancer = helper.lb;

    // 2. 创建新的 NameResolverListener
    NameResolverListenerImpl listener = new NameResolverListener
    Impl(helper);
    try {
        nameResolver.start(listener);
    } catch (Throwable t) {
        listener.onError(Status.fromThrowable(t));
    }
}
```

cancelIdleTimer()的实现

```
private void cancelIdleTimer() {
    if (idleModeTimerFuture != null) {
        // 取消future
        idleModeTimerFuture.cancel(false);
        // 设置timer为取消
        idleModeTimer.cancelled = true;
        // 然后都设置为null
        idleModeTimerFuture = null;
        idleModeTimer = null;
    }
}
```

rescheduleIdleTimer()的实现

```
private void rescheduleIdleTimer() {
    if (idleTimeoutMillis == IDLE_TIMEOUT_MILLIS_DISABLE) {
        // 这个检查控制着 空闲模式 的开启和关闭，具体分析见后面
        return;
    }
    // 取消现有的可能的timer，主要这个方法执行后 idleModeTimer 会被设置为null
    cancelIdleTimer();
    // 重新再构造一个timer
    idleModeTimer = new IdleModeTimer();
    // 重新再安排这个timer的执行
    idleModeTimerFuture = scheduledExecutor.schedule(
        new LogExceptionRunnable(new Runnable() {
            @Override
            public void run() {
                channelExecutor.executeLater(idleModeTimer).drain();
            }
        }),
        idleTimeoutMillis, TimeUnit.MILLISECONDS);
}
```

控制空闲模式的开启

上面可以看到空闲模式的开启由属性 `idleTimeoutMillis` 控制，具体看这个属性的使用：

```
static final long IDLE_TIMEOUT_MILLIS_DISABLE = -1;

// final属性
private final long idleTimeoutMillis;

ManagedChannelImpl(String target, ....., long idleTimeoutMillis
, .....) {
    .....
    if (idleTimeoutMillis == IDLE_TIMEOUT_MILLIS_DISABLE) {
        this.idleTimeoutMillis = idleTimeoutMillis;
    } else {
        checkArgument(idleTimeoutMillis >= AbstractManagedChannelImplBuilder.IDLE_MODE_MIN_TIMEOUT_MILLIS, "invalid idleTimeoutMillis %s", idleTimeoutMillis);
        this.idleTimeoutMillis = idleTimeoutMillis;
    }
    // 有效性检查，要开启就必须 > 0，要关闭就设置为 IDLE_TIMEOUT_MILLIS_DISABLE
    checkArgument(idleTimeoutMillis > 0 || idleTimeoutMillis == IDLE_TIMEOUT_MILLIS_DISABLE, "invalid idleTimeoutMillis %s", idleTimeoutMillis);
    // 构造函数中赋值
    this.idleTimeoutMillis = idleTimeoutMillis;
}
```

再往上追，控制权在构造 `ManagedChannelImpl` 时，只有一个调用的地方，`AbstractManagedChannelImplBuilder` 中的 `build()` 方法：

```
public ManagedChannelImpl build() {
    .....
    return new ManagedChannelImpl(....., idleTimeoutMillis, .....
.);
}
```

属性 `idleTimeoutMillis` 在 `build()` 时传入，而这个属性的设置是通过 `idleTimeout()` 方法：

```
// 默认值为 IDLE_TIMEOUT_MILLIS_DISABLE
// 因此如果不调用 idleTimeout()，默认是不会开启空闲模式的
private long idleTimeoutMillis = IDLE_TIMEOUT_MILLIS_DISABLE;

// 最大的空闲超时时间，比这个还大则将禁用空闲模式
// 最大空闲30天，也够夸张的
static final long IDLE_MODE_MAX_TIMEOUT_DAYS = 30;

// 默认超时时间
static final long IDLE_MODE_DEFAULT_TIMEOUT_MILLIS = TimeUnit.MINUTES.toMillis(30);

// 最小空闲时间为1秒，如果设置的比这个还短则会设置为1秒
static final long IDLE_MODE_MIN_TIMEOUT_MILLIS = TimeUnit.SECONDS.toMillis(1);

public final T idleTimeout(long value, TimeUnit unit) {
    // 检查必须 > 0，
    checkArgument(value > 0, "idle timeout is %s, but must be positive", value);
    // We convert to the largest unit to avoid overflow
    if (unit.toDays(value) >= IDLE_MODE_MAX_TIMEOUT_DAYS) {
        // 检查如果超过最大值，则禁用空闲模式
        this.idleTimeoutMillis = ManagedChannelImpl.IDLE_TIMEOUT_MILLIS_DISABLE;
    } else {
        // 转为毫秒单位，如果比最小容许值(1秒)还小则取最小容许值
        this.idleTimeoutMillis = Math.max(unit.toMillis(value), IDLE_MODE_MIN_TIMEOUT_MILLIS);
    }
    return thisT();
}
```

工作方式(深入)

前面发现空闲模式的进入和退出是由 InUseStateAggregator 来进行控制的，而 InUseStateAggregator 的工作方式请见下一节。

InUseStateAggregator 的工作方式

上一节发现空闲模式的进入和退出是由 InUseStateAggregator 来进行控制的，继续看 InUseStateAggregator 是如何工作的。

类InUseStateAggregator

InUseStateAggregator，字面意思是"正在使用的状态聚合器"，javadoc如此描述：

Aggregates the in-use state of a set of objects. 聚合一个对象集合的正在使用的状态

这是一个grpc的内部抽象类：

```
package io.grpc.internal;
abstract class InUseStateAggregator<T> {}
```

实现原理

类InUseStateAggregator的实现原理很直白，就是每个要使用这个聚合器的调用者都要存进来一个对象(就是范型T)，然后用完之后再取出来，这样就可以通过保存对象的数量变化判断开始使用(从无到有)或者已经不再使用(从有到无):

```

private final HashSet<T> inUseObjects = new HashSet<T>();

final void updateObjectInUse(T object, boolean inUse) {
    synchronized (getLock()) {
        // 记录更新前的原始数量
        int origSize = inUseObjects.size();
        if (inUse) {
            // inUse为true，表示增加正在使用的状态，保存对象
            inUseObjects.add(object);
            if (origSize == 0) {
                // 如果原始数量为0，表示从无到有，此时回调抽象方法 handleInUse
                handleInUse();
            }
        } else {
            // inUse为false，表示检查正在使用的状态，删除对象
            boolean removed = inUseObjects.remove(object);
            if (removed && origSize == 1) {
                // 如果删除成功并且原始使用为1，表示从有到无，此时回调抽象方法 h
                handleNotInUse();
            }
        }
    }
}

```

isInUse() 函数通过检查是否有保存的对象来判断是否正在使用：

```

final boolean isInUse() {
    synchronized (getLock()) {
        return !inUseObjects.isEmpty();
    }
}

```

状态变更回调

当状态变更时，回调这两个抽象方法：

1. `handleInUse()`：当被聚合的使用中的状态被修改为 `true` 时调用，这意味着至少一个对象正在使用中。

```
abstract void handleInUse();
```

2. `handleNotInUse()`：当被聚合的使用中的状态被修改为 `false` 时调用，这意味着没有对象正在使用中。

```
abstract void handleNotInUse();
```

ManagedChannellImpl中的实现

回顾 InUseStateAggregator 定义

回顾 ManagedChannellImpl 中对 inUseStateAggregator 的定义：

```
final InUseStateAggregator<Object> inUseStateAggregator =
    new InUseStateAggregator<Object>() {
        .....
        void handleInUse() {
            // 当状态变更为使用中时，退出空闲模式
            exitIdleMode();
        }

        void handleNotInUse() {
            // 当状态变更为没有人使用时，重新安排空闲timer以便在稍后进入空闲模式
            rescheduleIdleTimer();
        }
    };
```

调用 InUseStateAggregator

在ManagedChannellImpl中，当 transport 状态变更时就会调用 InUseStateAggregator：

```
final TransportManager<ClientTransport> tm = new TransportManager<ClientTransport>() {
    .....
    ts = new TransportSet(....., new TransportSet.Callback() {
        .....
        public void onInUse(TransportSet ts) {
            // 当 transportset 的状态变更为 InUse 时
            inUseStateAggregator.updateObjectInUse(ts, true);
        }

        public void onNotInUse(TransportSet ts) {
            // 当 transportset 的状态变更为 NotInUse 时
            inUseStateAggregator.updateObjectInUse(ts, false);
        }
    }
}
```

还有 InterimTransportImpl:

```
private class InterimTransportImpl implements InterimTransport<ClientTransport> {
    public void transportTerminated() {
        .....
        inUseStateAggregator.updateObjectInUse(delayedTransport, false);
    }

    @Override public void transportInUse(boolean inUse) {
        inUseStateAggregator.updateObjectInUse(delayedTransport, inUse);
    }
}
```

注：Transport 的详细内容不再继续展开了。

Name Resolver

构建 Name Resolver

ManagedChannelImpl中和 Name Resolver 相关的属性：

```
// 匹配这个正则表达式意味着 target 字符串是一个 URI target，或者至少打算
// 成为一个
// URI target 必须是一个absolute hierarchical URI
// 来自 RFC 2396: scheme = alpha *( alpha | digit | "+" | "-" | "
// ." )
static final Pattern URI_PATTERN = Pattern.compile("[a-zA-Z][a-zA-Z0-9+.-]*:/.*");

private final String target;
private final NameResolver.Factory nameResolverFactory;
private final Attributes nameResolverParams;

// 绝不为空。必须在lock之下修改。
// 这里不能是final，因为后面会重新设值，但是绝不会设置为null
private NameResolver nameResolver;
```

ManagedChannelImpl()的构造函数，传入
target/nameResolverFactory/nameResolverParams：

```

ManagedChannelImpl(String target, .....
    NameResolver.Factory nameResolverFactory, Attributes nameR
esolverParams,.....) {
    this.target = checkNotNull(target, "target");
    this.nameResolverFactory = checkNotNull(nameResolverFactory,
"nameResolverFactory");
    this.nameResolverParams = checkNotNull(nameResolverParams, "
nameResolverParams");
    this.nameResolver = getNameResolver(target, nameResolverFact
ory, nameResolverParams);
    .....
}

```

getNameResolver() 方法根据这三个参数构建 NameResolver :

```

static NameResolver getNameResolver(String target, NameResolver.
Factory nameResolverFactory, Attributes nameResolverParams) {
    //查找NameResolver。尝试使用 target 字符串作为 URI。如果失败，尝试
添加前缀
    // "dns:///".
    URI targetUri = null;
    StringBuilder uriSyntaxErrors = new StringBuilder();
    try {
        targetUri = new URI(target);
        // 对于 "localhost:8080" 这将会导致 newNameResolver() 方法返回
null
        // 因为 "localhost" 被作为 scheme 解析。
        // 将会转入下一个分支并尝试 "dns:///localhost:8080"
    } catch (URISyntaxException e) {
        // 可以发生在类似 "[::1]:1234" or 127.0.0.1:1234 这样的ip地址
        uriSyntaxErrors.append(e.getMessage());
    }
    if (targetUri != null) {
        // 如果 target 是有效的URI
        NameResolver resolver = nameResolverFactory.newNameResolve
r(targetUri, nameResolverParams);
        if (resolver != null) {
            return resolver;
        }
    }
}

```

```

    }
    // "foo.googleapis.com:8080" 导致 resolver 为 null，因为 "foo
    ..googleapis.com" 是一个没有映射的 scheme。简单失败并将尝试"dns:///foo.g
    oogleapis.com:8080"
    }

    // 如果走到这里，表明 targetUri 不能使用
    if (!URI_PATTERN.matcher(target).matches()) {
        // 如果格式都不匹配，说明看上去不像是一个 URI target。可能是 autho
        rity 字符串。
        // 尝试从factory中获取默认 scheme
        try {
            targetUri = new URI(nameResolverFactory.getDefaultScheme
            (), "", "/" + target, null);
        } catch (URISyntaxException e) {
            // Should not be possible.
            throw new IllegalArgumentException(e);
        }
        if (targetUri != null) {
            // 尝试加了默认 scheme 之后的URI
            NameResolver resolver = nameResolverFactory.newNameResol
            ver(targetUri, nameResolverParams);
            if (resolver != null) {
                return resolver;
            }
        }
    }
    // 最后如果还是不成，只能报错了
    throw new IllegalArgumentException(String.format(
        "cannot find a NameResolver for %s%s",
        target, uriSyntaxErrors.length() > 0 ? " (" + uriSyntaxE
        rrors + ")" : ""));
}

```

构建失败会直接抛异常退出。

Name Resolver 的使用

在 TransportSet.Callback 中被调用：


```

final TransportManager<ClientTransport> tm = new TransportManager<ClientTransport>() {
    public ClientTransport getTransport(final EquivalentAddressGroup addressGroup) {
        .....
        ts = new TransportSet(.....,
            new TransportSet.Callback() {
                public void onAllAddressesFailed() {
                    // 所有地址失败时刷新
                    nameResolver.refresh();
                }

                public void onConnectionClosedByServer(Status status)
            {
                // 服务器端关闭连接时刷新
                nameResolver.refresh();
            }
            .....
        }
    }
}

```

NameResolver.Listener

NameResolver通过 NameResolver.Listener 来通知变化，这里的实现是类 NameResolverListenerImpl：

```

private class NameResolverListenerImpl implements NameResolver.Listener {
    final LoadBalancer<ClientTransport> balancer;

    NameResolverListenerImpl(LoadBalancer<ClientTransport> balancer) {
        this.balancer = balancer;
    }

    @Override
    public void onUpdate(List< ? extends List<ResolvedServerInfo>> servers, Attributes config) {
        if (serversAreEmpty(servers)) {

```

```
// 如果更新的服务列表为空，表明没有可用的服务器了
onError(Status.UNAVAILABLE.withDescription("NameResolver
returned an empty list"));
} else {
    // 如果更新的服务列表不为空，通知balancer
    try {
        balancer.handleResolvedAddresses(servers, config);
    } catch (Throwable e) {
        // 这必然是一个bug！将异常推回给 LoadBalancer，希望LoadBal
        ancer可以提升给应用
        balancer.handleNameResolutionError(Status.INTERNAL.wit
        hCause(e)
            .withDescription("Thrown from handleResolvedAddres
            ses(): " + e));
    }
}

@Override
public void onError(Status error) {
    checkArgument(!error.isOk(), "the error status must not be
    OK");
    // 通知 balancer 解析错误
    balancer.handleNameResolutionError(error);
}
}
```

name resolver 的 start()的触发

而类NameResolverListenerImpl只有一个地方使用，在方法 exitIdleMode() 中：

```

LoadBalancer<ClientTransport> exitIdleMode() {
    synchronized (lock) {
        .....
        if (loadBalancer != null) {
            // 如果loadBalancer不为空就直接return
            // 意味着如果想继续后面的代码，就必须loadBalancer为空才行
            return loadBalancer;
        }
        balancer = loadBalancerFactory.newLoadBalancer(nameResolver.getServiceAuthority(), tm);
        this.loadBalancer = balancer;
        resolver = this.nameResolver;
    }
    class NameResolverStartTask implements Runnable {
        @Override
        public void run() {
            // 这将在 LoadBalancer 和 NameResolver 中触发一些不平凡的工作
            // 不想在lock里面做(因此封装成task扔给scheduledExecutor)
            resolver.start(new NameResolverListenerImpl(balancer));
        }
    }

    scheduledExecutor.execute(new NameResolverStartTask());
    return balancer;
}

```

在 NameResolverStartTask 中才开始调用 name resolver 的 start() 方法启动 name resolver 的工作。而搜索中发现，这里是 name resolver 的 start() 方法的唯一一个使用的地方。也就是说，这是 name resolver 开始工作的唯一入口。

因此，name resolver 要开始工作，需要两个条件：

1. exitIdleMode() 方法被调用
2. loadBalancer 属性为null

exitIdleMode() 方法有两个被调用的地方：

1. inUseStateAggregator中，当发现 Channel 的状态转为使用中时：

```

    final InUseStateAggregator<Object> inUseStateAggregator = n
ew InUseStateAggregator<Object>() {
    .....
    void handleInUse() {
        exitIdleMode();
    }
}

```

2. ClientTransportProvider的get()方法，通过exitIdleMode()方法获取balancer，然后通过balancer获取ClientTransport：

```

    private final ClientTransportProvider transportProvider = n
ew ClientTransportProvider() {
        @Override
        public ClientTransport get(CallOptions callOptions) {
            LoadBalancer<ClientTransport> balancer = exitIdleMode
();
            if (balancer == null) {
                return SHUTDOWN_TRANSPORT;
            }
            return balancer.pickTransport(callOptions.getAffinity
());
        }
    };

```

这里返回的 ClientTransport 在RealChannel的ClientCall()方法中使用,被传递给新创建的ClientCallImpl：

```

    private class RealChannel extends Channel {

        public <ReqT, RespT> ClientCall<ReqT, RespT> newCall(Method
Descriptor<ReqT, RespT> method, CallOptions callOptions) {
            .....
            return new ClientCallImpl<ReqT, RespT>(. . . . .
                transportProvider, . . . . .)
        }
    }

```

而ClientCallImpl中这个get()方法只有一个地方被调用，在ClientCallImpl的start()方法中：

```
public void start(final Listener<RespT> observer, Metadata
headers) {
    .....
    ClientTransport transport = clientTransportProvider.get
(callOptions);
}
```

name resolver 的总结

name resolver 的 start() 方法，也就是 name resolver 要开始解析name的这个工作，只有两种情况下开始：

1. 第一次RPC请求：此时要调用Channel的 newCall() 方法得到ClientCall的实例，然后调 ClientCall 的 start()方法，期间获取ClientTransport时激发一次 name resolver 的 start()
2. 如果开启了空闲模式：则在每次 Channel 从空闲模式退出，进入使用状态时，再激发一次 name resolver 的 start()

Load Balancer

```
private final LoadBalancer.Factory loadBalancerFactory;  
  
// channel处于空闲模式时 channel 为 null  
@Nullable  
private LoadBalancer<ClientTransport> loadBalancer;
```

Transport

```
private static final ClientTransport SHUTDOWN_TRANSPORT =  
    new FailingClientTransport(Status.UNAVAILABLE.withDescription(  
        "Channel is shutdown"));  
  
static final ClientTransport IDLE_MODE_TRANSPORT =  
    new FailingClientTransport(Status.INTERNAL.withDescription("Ch  
annel is in idle mode"));  
  
private final ClientTransportFactory transportFactory;
```

Executor

```
private final Executor executor;  
private final boolean usingSharedExecutor;
```


关闭

```
private boolean shutdown;  
private boolean shutdownNowed;  
private boolean terminated;
```

Channel Builder设计与代码实现

功能

Channel Builder 用于帮助创建 Channel 对象。

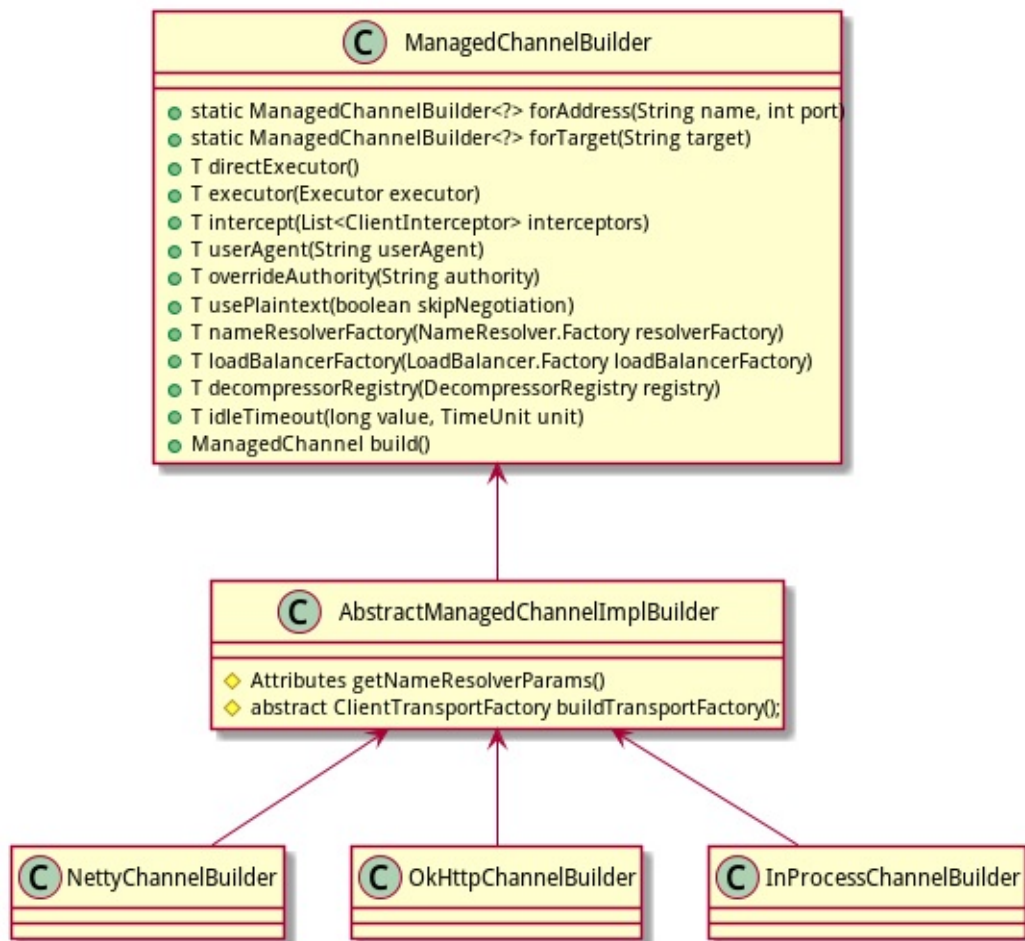
使用场景

标准的实现方式是:

```
NettyChannelBuilder builder = NettyChannelBuilder.forAddress("127.0.0.1", 1080);
builder.someMethod().....
ManagedChannel channel = builder.build();
```

继承结构

Channel Builder 的继承结构如下：



Channel Builder 有三个实现，对应于 netty/okjava/inprocess 三种主要的 Channel

。

类ManagedChannelBuilder

用于创建 ManagedChannel 实例的 builder.

类定义

```
package io.grpc;  
public abstract class ManagedChannelBuilder<T extends ManagedChannelBuilder<T>> {  
}
```

T 是这个builder的具体类型。

类方法

静态工厂方法

有两个静态方法用来实例。

ManagedChannelProvider 的使用方式和设计实现，不在这里展开，在下一节 [Channel Provider](#) 中再详细介绍。

forAddress()

```
public static ManagedChannelBuilder<?> forAddress(String name, int port) {  
    return ManagedChannelProvider.provider().builderForAddress(name, port);  
}
```

forTarget()

```
@ExperimentalApi
public static ManagedChannelBuilder<?> forTarget(String target)
{
    return ManagedChannelProvider.provider().builderForTarget(target);
}
```

用target字符串创建一个channel，可以是一个有效的命名解析器兼容(NameResolver-compliant)的URI，或者是一个 authority 字符串。

注：authority 是URI中的术语，URI的标准组成部分如 [scheme:] [//authority][path][?query][#fragment]，authority代表URI中的 [userinfo@]host[:port]，包括host(或者ip)和可选的port和userinfo。

命名解析器兼容(NameResolver-compliant)URI 是被作为URI定义的抽象层次(absolute hierarchical)URI。实例的URI：

- "dns:///foo.googleapis.com:8080"
- "dns:///foo.googleapis.com"
- "dns:///5B2001:db8:85a3:8d3:1319:8a2e:370:7348%5D:443"
- "dns://8.8.8.8/foo.googleapis.com:8080"
- "dns://8.8.8.8/foo.googleapis.com"
- "zookeeper://zk.example.com:9900/example_service"

authority字符串将被转换为一个命名解析器兼容的URI，使用"dns"作为scheme，没有authority，而且用合适转义之后的原始authority字符串作为它的path：

- "localhost"
- "127.0.0.1"
- "localhost:8080"
- "foo.googleapis.com:8080"
- "127.0.0.1:8080"
- "[2001:db8:85a3:8d3:1319:8a2e:370:7348]"
- "[2001:db8:85a3:8d3:1319:8a2e:370:7348]:443"

注：以上内容来自javadoc上该方法的注释说明。

实例方法

directExecutor()

```
public abstract T directExecutor();
```

直接在传输的线程中执行应用代码。

取决于底层传输，使用一个 **direct executor** 可能引发重大的性能提升。当然，它也要求应用在任何情况下不阻塞。

调用这个方法在语义上等同于调用 **executor(Executor)** 并传递一个 **direct executor**。但是，这个方式更合适因为它可以容许传输层执行特别的优化。

注：考虑实际测试一下，看是否真的可以得到重大的性能提升。

executor()

```
public abstract T executor(Executor executor);
```

提供一个自定义**executor**。

这是一个可选参数。如果在**channel**构建时没有提供**executor**，**builder**将使用一个静态缓存的线程池。

channel不会承担给定 **executor** 的所有者职责。当需要时，关闭 **executor** 是调用者的责任。

注：这意味着我们作为调用者需要自己管理好 **executor**。

intercept()

```
public abstract T intercept(List<ClientInterceptor> interceptors)  
;
```

添加拦截器，被**channel**执行它的实际工作前被调用。功能上等同于使用 **ClientInterceptors.intercept(Channel, List)**，但是依然可以访问原始的 **ManagedChannel**。

```
public abstract T intercept(ClientInterceptor... interceptors);
```

添加拦截器，被channel执行它的实际工作前被调用。功能上等同于使用 ClientInterceptors.intercept(Channel, ClientInterceptor...)，但是依然可以访问原始的ManagedChannel。

userAgent()

```
public abstract T userAgent(String userAgent);
```

为应用提供一个定制的 User-Agent。

这是一个可选参数。如果提供，给定的agent将使用grpc User-Agent作为前缀。

注：原文 the given agent will be prepended by the grpc User-Agent. 硬是没有看懂到底哪个作为前缀？等后面查代码。

overrideAuthority()

```
@ExperimentalApi(value="primarily for testing")  
public abstract T overrideAuthority(String authority)
```

覆盖和TLS和HTTP 虚拟主机服务一起使用的authority。它不会改变实际连接到的主机。通常是以host:port的形式。

应该仅用于测试。

usePlaintext()

```
@ExperimentalApi("primarily for testing")  
public abstract T usePlaintext(boolean skipNegotiation);
```

使用 plaintext 连接到服务器。默认将使用加密连接机制如 TLS。

应该仅用于测试或者用于那些API使用或者数据交换并不敏感的API。

参数 `skipNegotiation true`，如果预先知道终端支持plaintext; `false` 如果 plaintext 的使用必须协商。

nameResolverFactory()

```
@ExperimentalApi
public abstract T nameResolverFactory(NameResolver.Factory resolverFactory);
```

为channel提供一个定制的 `NameResolver.Factory`。如果这个方法没有被调用，builder将在全局解析器注册(global resolver registry)中为提供的目标寻找一个工厂。

loadBalancerFactory()

```
@ExperimentalApi
public abstract T loadBalancerFactory(LoadBalancer.Factory loadBalancerFactory);
```

为channel提供一个定制的 `LoadBalancer.Factory`。

如果这个方法没有被调用，builder将为这个 channel 使用 `SimpleLoadBalancerFactory`。

decompressorRegistry()

```
@ExperimentalApi
public abstract T decompressorRegistry(DecompressorRegistry registry);
```

设置用于在channel中使用的解压缩注册器。这是一个高级API调用，不应该被使用，除非你正在使用定制化的消息编码。默认支持的解压缩在 `DecompressorRegistry.getDefaultInstance` 中。

compressorRegistry()

```
@ExperimentalApi
public abstract T compressorRegistry(CompressorRegistry registry)
;
```

设置用于在channel中使用的压缩注册器。这是一个高级API调用，不应该被使用，除非你正在使用定制化的消息编码。默认支持的解压缩在DecompressorRegistry.getDefaultInstance 中。

idleTimeout()

```
@ExperimentalApi
public abstract T idleTimeout(long value, TimeUnit unit);
```

设置在进入空闲模式前的没有RPC的期限。

在空闲模式中，channel会关闭所有连接，NameResolver 和 LoadBalancer。新的RPC将把channel带出空闲模式。channel以空闲模式开始。

默认，在离开初始化空闲模式 channel 将从不会进入空闲模式

build()

使用给定参数来构建一个channel

```
@ExperimentalApi
public abstract ManagedChannel build();
```

类AbstractManagedChannelImplBuilder

类AbstractManagedChannelImplBuilder 是 channel builder的基类。

类定义

```
package io.grpc.internal;
public abstract class AbstractManagedChannelImplBuilder
    <T extends AbstractManagedChannelImplBuilder<T>> extends
    ManagedChannelBuilder<T> {}
```

注：package从 ManagedChannelBuilder 的 io.grpc 变成 io.grpc.internal 了。

类属性和方法实现

executor

```
@Nullable
private Executor executor;

@Override
public final T directExecutor() {
    return executor(MoreExecutors.directExecutor());
}

@Override
public final T executor(Executor executor) {
    this.executor = executor;
    return thisT();
}
```

executor(Executor executor)方法只是一个简单赋值，而directExecutor()则调用MoreExecutors.directExecutor() 方法得到 direct executor。

executor 在 build() 方法中被传递给新创建的 ManagedChannelImpl() 实例。

访问地址相关

和访问地址相关的三个属性：

1. target
2. directServerAddress
3. nameResolverFactory

```
private final String target;  
@Nullable  
private final SocketAddress directServerAddress;  
@Nullable  
private NameResolver.Factory nameResolverFactory;
```

使用target来构造实例：

```
protected AbstractManagedChannelImplBuilder(String target) {  
    this.target = Preconditions.checkNotNull(target);  
    // directServerAddress 设置为null  
    this.directServerAddress = null;  
}
```

使用directServerAddress来构造实例：

```
private static final String DIRECT_ADDRESS_SCHEME = "directaddress";
protected AbstractManagedChannelImplBuilder(SocketAddress directServerAddress, String authority) {
    // target被设置为 directaddress:///directServerAddress
    this.target = makeTargetStringForDirectAddress(directServerAddress);
    this.directServerAddress = directServerAddress;
    // nameResolverFactory 使用 DirectAddressNameResolverFactory
    this.nameResolverFactory = new DirectAddressNameResolverFactory(directServerAddress, authority);
}

@VisibleForTesting
static String makeTargetStringForDirectAddress(SocketAddress address) {
    try {
        // "directaddress:///address"
        return new URI(DIRECT_ADDRESS_SCHEME, "", "/" + address, null).toString();
    } catch (URISyntaxException e) {
        // It should not happen.
        throw new RuntimeException(e);
    }
}
```

设置nameResolverFactory的函数，注意NameResolverFactory和directServerAddress是互斥的：

```
@Override
public final T nameResolverFactory(NameResolver.Factory resolver
Factory) {
    // 如果设置directServerAddress，就不能再使用NameResolverFactory
    Preconditions.checkState(directServerAddress == null,
        "directServerAddress is set (%s), which forbids the use
of NameResolverFactory",
        directServerAddress);
    this.nameResolverFactory = resolverFactory;
    return thisT();
}
```

target 和 nameResolverFactory 在 build() 方法中被传递给新创建的 ManagedChannelImpl() 实例。

user agent

```
@Override
@Nullable
private String userAgent;

public final T userAgent(String userAgent) {
    this.userAgent = userAgent;
    return thisT();
}
```

user agent 只是做了一个简单赋值，然后在 build() 方法中被传递给新创建的 ManagedChannelImpl() 实例。

authorityOverride

```
@Nullable
private String authorityOverride;

@Override
public final T overrideAuthority(String authority) {
    this.authorityOverride = checkAuthority(authority);
    return thisT();
}
```

简单做了个赋值，然后在build()方法中，在构建AuthorityOverridingTransportFactory时使用：

```
@Override
public ManagedChannelImpl build() {
    ClientTransportFactory transportFactory = buildTransportFactory();
    if (authorityOverride != null) {
        transportFactory = new AuthorityOverridingTransportFactory(
            transportFactory, authorityOverride);
    }
    .....
    return new ManagedChannelImpl(
        ....., transportFactory, .....);
}
```

通过buildTransportFactory()方法得到ClientTransportFactory，然后传递给ManagedChannelImpl的构造函数。为了覆盖原来的Authority，实现了一个AuthorityOverridingTransportFactory内部类，以装饰模式包裹了一个ClientTransportFactory的实例，然后将请求都代理给这个包装的ClientTransportFactory实例：

```
private static class AuthorityOverridingTransportFactory implements ClientTransportFactory {
    final ClientTransportFactory factory;
    final String authorityOverride;

    AuthorityOverridingTransportFactory(
        ClientTransportFactory factory, String authorityOverride
    ) {
        this.factory = Preconditions.checkNotNull(factory, "factory should not be null");
        this.authorityOverride = Preconditions.checkNotNull(
            authorityOverride, "authorityOverride should not be null"
        );
    }

    @Override
    public ConnectionClientTransport newClientTransport(SocketAddress serverAddress,
        String authority, @Nullable String userAgent) {
        // 在这里做覆盖，用authorityOverride覆盖原有的authority
        return factory.newClientTransport(serverAddress, authorityOverride, userAgent);
    }
    .....
}
```

在newClientTransport()方法中，前面传递进来的 authorityOverride 派上用场了。

注：之前AuthorityOverridingTransportFactory的实现有点小问题，我提交了一个pull request给grpc，后来被采纳，现在这里的代码已经合并到master，看着真亲切 :)： <https://github.com/grpc/grpc-java/pull/1666>

nameResolverFactory

```
@Nullable
private NameResolver.Factory nameResolverFactory;

protected AbstractManagedChannelImplBuilder(SocketAddress direct
ServerAddress, String authority) {
    .....
    this.nameResolverFactory = new DirectAddressNameResolverFact
ory(directServerAddress, authority);
}

@Override
public final T nameResolverFactory(NameResolver.Factory resolver
Factory) {
    Preconditions.checkState(directServerAddress == null,
        "directServerAddress is set (%s), which forbids the use
of NameResolverFactory",
        directServerAddress);
    this.nameResolverFactory = resolverFactory;
    return thisT();
}
```

如前所述，如果设置了directServerAddress，则nameResolverFactory自动设置为DirectAddressNameResolverFactory。而且不能再设置nameResolverFactory。

nameResolverFactory在build()方法中使用，有一个null的检查，如果nameResolverFactory有设置则使用nameResolverFactory，否则使用默认的NameResolverProvider.asFactory()：


```
@Override
public ManagedChannelImpl build() {
    .....
    NameResolver.Factory nameResolverFactory = this.nameResolver
Factory;
    if (nameResolverFactory == null) {
        nameResolverFactory = NameResolverProvider.asFactory();
    }
    return new ManagedChannelImpl(
        .....,
        nameResolverFactory,
        getNameResolverParams(),
        .....);
}
```

注：这里的NameResolverProvider.asFactory()看不懂.....

额外的getNameResolverParams()用来给子类使用(以override的方式)，可以传递更多参数给 NameResolver.Factory.newNameResolver() 方法。默认实现只是返回一个Attributes.EMPTY

```
protected Attributes getNameResolverParams() {
    return Attributes.EMPTY;
}
```

loadBalancerFactory

```

@Nullable
private LoadBalancer.Factory loadBalancerFactory;

@Override
public final T loadBalancerFactory(LoadBalancer.Factory loadBalancerFactory) {
    Preconditions.checkState(directServerAddress == null,
        "directServerAddress is set (%s), which forbids the use of LoadBalancerFactory",
        directServerAddress);
    this.loadBalancerFactory = loadBalancerFactory;
    return thisT();
}

public ManagedChannelImpl build() {
    return new ManagedChannelImpl(
        .....,
        firstNonNull(loadBalancerFactory, DummyLoadBalancerFactory.getInstance()),
        .....);
}

```

和 `nameResolverFactory` 的使用非常类似，同样是 `directServerAddress` 设置后不能再使用，使用的方式也是在 `build()` 中检查是否有设置，如果没有设置则默认使用 `DummyLoadBalancerFactory`。

decompressorRegistry 和 compressorRegistry

```

public ManagedChannelImpl build() {
    return new ManagedChannelImpl(
        .....,
        firstNonNull(decompressorRegistry, DecompressorRegistry.getDefaultInstance()),
        firstNonNull(compressorRegistry, CompressorRegistry.getDefaultInstance()),
        .....);
}

```

decompressorRegistry 和 compressorRegistry 基本就是个简单设置+ 在build()方法中传递给ManagedChannelImpl()，如果没有设置则用默认值。

关键方法

抽象方法 **buildTransportFactory()**

定义抽象方法 buildTransportFactory()，用于子类实现这个方法为这个channel 提供 ClientTransportFactory。这个方法只对Transport 的实现者有意义，而不应该被普通用户使用。

```
protected abstract ClientTransportFactory buildTransportFactory()  
;
```

返回的 ClientTransportFactory 将在 build() 方法中传递给 ManagedChannelImpl() 的构造函数。

类NettyChannelBuilder

在构建使用netty transport的channel的过程中，用来帮助简化d的builder。

类定义

```
@ExperimentalApi("https://github.com/grpc/grpc-java/issues/1784")
)
public class NettyChannelBuilder extends AbstractManagedChannelI
mplBuilder<NettyChannelBuilder> {}
```

注: 都1.0.0-pre2了居然还是 @ExperimentalApi.....

方法实现

最重要的buildTransportFactory()

```
@Override
protected ClientTransportFactory buildTransportFactory() {
    return new NettyTransportFactory(channelType, negotiationType,
    protocolNegotiator, sslContext, eventLoopGroup, flowControlWindow,
    maxMessageSize, maxHeaderListSize);
}
```

new了一个NettyTransportFactory实例，然后给了一堆参数。

下面看看各个参数的含义和传递/使用方式。

channelType

使用的channel type，默认使用NioSocketChannel。貌似一般也都是用这个了。

```
private Class<? extends Channel> channelType = NioSocketChannel.class;

public final NettyChannelBuilder channelType(Class<? extends Channel> channelType) {
    this.channelType = Preconditions.checkNotNull(channelType);
    return this;
}
```

negotiationType

协商类型，具体看 `io.grpc.netty.NegotiationType` 这个枚举的说明。

`NegotiationType` 用于定义启动HTTP/2的协商方式：

- TLS：使用 `TLS ALPN/NPN` 协商，假定是 SSL 连接
- PLAINTEXT_UPGRADE：使用 HTTP 升级协议为 plaintext(非SSL)从 HTTP/1.1 升级到 HTTP/2
- PLAINTEXT: 假设连接是 plaintext(非SSL) 而且远程端点直接支持HTTP2.而不需要升级

从实现代码上看，默认是TLS，然后通过方法设置：

```
private NegotiationType negotiationType = NegotiationType.TLS;

public final NettyChannelBuilder negotiationType(NegotiationType type) {
    negotiationType = type;
    return this;
}
```

此外还有一个`usePlaintext()`方法，指明使用 `Plaintext`，然后通过参数 `skipNegotiation` 来指定是否需要跳过协商过程。这个方法等同于调用 `negotiationType(PLAINTEXT)`或`negotiationType(PLAINTEXT_UPGRADE)`,取决于参数取值：

```
@Override
public final NettyChannelBuilder usePlaintext(boolean skipNegotiation) {
    if (skipNegotiation) {
        negotiationType(NegotiationType.PLAINTEXT);
    } else {
        negotiationType(NegotiationType.PLAINTEXT_UPGRADE);
    }
    return this;
}
```

从代码上看，只是一个简单的if判断，这个方法可以认为是 negotiationType() 方法的一个可读性稍好的版本，存在的价值只是为了方便和易读。

protocolNegotiator

设置使用的protocolNegotiator，如果不是null，则覆盖 negotiationType(NegotiationType) 或者 usePlaintext(boolean):

```
private ProtocolNegotiator protocolNegotiator;

@Internal
public final NettyChannelBuilder protocolNegotiator(
    @Nullable ProtocolNegotiator protocolNegotiator) {
    this.protocolNegotiator = protocolNegotiator;
    return this;
}
```

所谓覆盖，是这样实现的，在newClientTransport()方法中，先判断 protocolNegotiator：

```
public ManagedClientTransport newClientTransport(  
    SocketAddress serverAddress, String authority) {  
    .....  
    ProtocolNegotiator negotiator = protocolNegotiator != null ?  
    protocolNegotiator :  
        createProtocolNegotiator(authority, negotiationType, sslCo  
    ntext);  
    return newClientTransport(serverAddress, authority, negotiat  
    or);  
}
```

只有 protocolNegotiator 为null时，才会通过使用 authority / negotiationType / sslContext 参数来创建ProtocolNegotiator。如果不为null，则直接使用，无视上述几个参数。

sslContext

可以设置 SL/TLS 上下文来使用，替代系统默认。必须已经用 GrpcSslContexts 配置过，但是选项可以被覆盖。

这个参数只是简单赋值，在上面的newClientTransport()方法中使用，注意同样的，如果protocolNegotiator被设置(不为null)，则这个sslContext将被忽略。

eventLoopGroup

可以提供一个 EventGroupLoop 给netty传输使用。

这是一个可选参数。在channel构建时如果用户没有提供 EventGroupLoop ，builder将使用默认，而这个默认是静态的。注意 channel 不会为给定的 EventGroupLoop 负责，在需要时调用者有责任关闭它。

```

@Nullable
private EventLoopGroup eventLoopGroup;

public final NettyChannelBuilder eventLoopGroup(@Nullable EventL
oopGroup eventLoopGroup) {
    this.eventLoopGroup = eventLoopGroup;
    return this;
}

```

所谓使用静态的默认，代码实现是这样的：

```

usingSharedGroup = group == null;
if (usingSharedGroup) {
    // The group was unspecified, using the shared group.
    this.group = SharedResourceHolder.get(Utils.DEFAULT_WORKER_E
VENT_LOOP_GROUP);
} else {
    this.group = group;
}

```

如果没有设置，则取

SharedResourceHolder.get(Utils.DEFAULT_WORKER_EVENT_LOOP_GROUP)，而DEFAULT_WORKER_EVENT_LOOP_GROUP是这样定义的：

```

class Utils {
    public static final Resource<EventLoopGroup> DEFAULT_BOSS_EVEN
T_LOOP_GROUP =
        new DefaultEventLoopGroupResource(1, "grpc-default-boss-EL
G");
    public static final Resource<EventLoopGroup> DEFAULT_WORKER_EV
ENT_LOOP_GROUP =
        new DefaultEventLoopGroupResource(0, "grpc-default-worker-
ELG");
}

```

构造函数中 worker的 numEventLoops 参数设置为0, 而另一个 boss 的 numEventLoops 参数设置为1。这会影响两个EventLoopGroup(可以理解为netty的线程池)的线程数，具体代码如下：


```
@Override
public EventLoopGroup create() {
    .....
    int parallelism = numEventLoops == 0
        ? Runtime.getRuntime().availableProcessors() * 2 : numEventLoops;
    return new NioEventLoopGroup(parallelism, threadFactory);
}
```

可以看到默认的 DEFAULT_WORKER_EVENT_LOOP_GROUP 的线程数为当前系统cpu数量的两倍。

注：这个 SharedResourceHolder 的实现挺有意思，可以拿来在需要时参考一下，甚至直接搬过来用。

flowControlWindow / maxMessageSize / maxHeaderListSize

这三个参数只是简单赋值和传递，然后给出了缺省的默认值：

```
private int flowControlWindow = DEFAULT_FLOW_CONTROL_WINDOW;
private int maxMessageSize = DEFAULT_MAX_MESSAGE_SIZE;
private int maxHeaderListSize = GrpcUtil.DEFAULT_MAX_HEADER_LIST_SIZE;
```

Channel Provider设计与代码实现

功能

Channel Provider 的功能在于帮助创建合适的 ManagedChannelBuilder。

所谓合适，是指目前有多套 Channel 的实现，典型如 netty 和 okhttp，不排除未来加入其他实现的可能。因此，如何选择哪套实现就是一个需要特别考虑的问题。

Channel Provider 的设计目标是解耦这个事情，不使用配置，hard code等方式，而是将细节交给 Channel Provider 的具体实现。

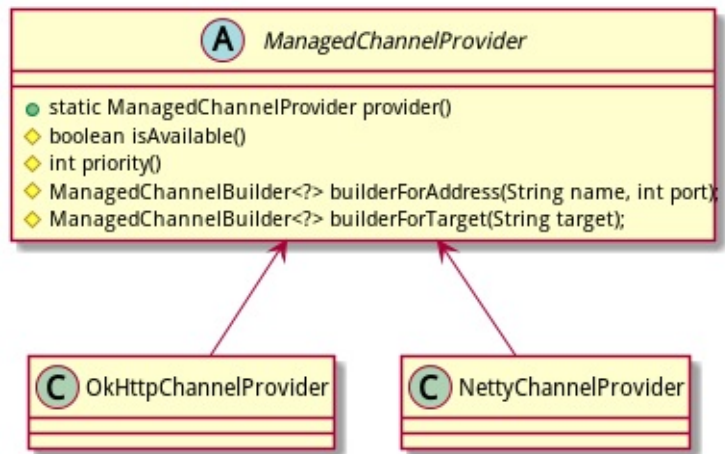
使用场景

在 ManagedChannelBuilder 中这样调用 ManagedChannelProvider：

```
public abstract class ManagedChannelBuilder<T extends ManagedChannelBuilder<T>> {  
    public static ManagedChannelBuilder<?> forAddress(String name,  
int port) {  
        return ManagedChannelProvider.provider().builderForAddress(name,  
port);  
    }  
    public static ManagedChannelBuilder<?> forTarget(String target)  
    ) {  
        return ManagedChannelProvider.provider().builderForTarget(target);  
    }  
    .....  
}
```

其中 provider() 静态方法会根据实际情况选择一套可用的方案，然后 builderForAddress()方法和 forTarget() 方法会创建对应的ManagedChannelBuilder (基于okjava或者netty)。

继承结构



结构简单，一个抽象基类 `ManagedChannelProvider`，然后 `okjava` 和 `netty` 各实现了一个子类。

具体实现看后面的代码分析。

类ManagedChannelProvider

ManagedChannelProvider 是 managed channel 的提供者，用于 transport 的不可知消费。

ManagedChannelProvider 的实现类不能抛出(异常)。如果抛出了，可能打断类装载。如果因为实现特有的原因会发生异常，实现应该优雅的处理异常并从 `isAvailable()` 方法返回 `false`。

注：以上说明，来自 ManagedChannelProvider 的 Javadoc。

类定义

```
package io.grpc;

@Internal
public abstract class ManagedChannelProvider {}
```

静态初始化

上面Javadoc的描述中提到的"类不能抛出异常，否则会打断类装载"，说的应该就是这个静态初始化的操作：

```
private static final ManagedChannelProvider provider
    = load(getCorrectClassLoader());
```

`getCorrectClassLoader()`方法先获取正确的ClassLoader:

```
private static ClassLoader getCorrectClassLoader() {  
    if (isAndroid()) {  
        //对于android平台，特殊一些  
        return ManagedChannelProvider.class.getClassLoader();  
    }  
    //其他情况，默认都是取当前线程的Context ClassLoader  
    return Thread.currentThread().getContextClassLoader();  
}
```

然后load()方法装载并选择ManagedChannelProvider的具体实现，这里有三个步骤：

1. 装载所有可能的候选者
2. 排除掉不可用的候选者
3. 选择优先级最高的候选者

load()方法的具体代码：

```
static ManagedChannelProvider load(ClassLoader classLoader) {
    Iterable<ManagedChannelProvider> candidates;
    if (isAndroid()) {
        // 对于android平台，直接hard code候选者
        candidates = getCandidatesViaHardCoded(classLoader);
    } else {
        // 其他情况，通过JDK的ServiceLoader装载候选者
        candidates = getCandidatesViaServiceLoader(classLoader);
    }
    List<ManagedChannelProvider> list = new ArrayList<ManagedChannelProvider>();
    for (ManagedChannelProvider current : candidates) {
        // 排除掉不可用的候选者
        if (!current.isAvailable()) {
            continue;
        }
        list.add(current);
    }
    if (list.isEmpty()) {
        // 如果为空返回null
        return null;
    } else {
        // 返回优先级最高的候选者
        return Collections.max(list, new Comparator<ManagedChannelProvider>() {
            @Override
            public int compare(ManagedChannelProvider f1, ManagedChannelProvider f2) {
                return f1.priority() - f2.priority();
            }
        });
    }
}
```

逻辑很清晰，继续看其中的两个细节，看候选者是如何被装载出来的。

1. android平台：hard code 两个可能的实现 okhttp 和 netty，如果类装载就只能忽略

```

    public static Iterable<ManagedChannelProvider> getCandidatesViaHardCoded(
        ClassLoader classLoader) {
        List<ManagedChannelProvider> list = new ArrayList<ManagedChannelProvider>();
        try {
            list.add(create(Class.forName("io.grpc.okhttp.OkHttpChannelProvider", true, classLoader)));
        } catch (ClassNotFoundException ex) {
            // ignore
        }
        try {
            list.add(create(Class.forName("io.grpc.netty.NettyChannelProvider", true, classLoader)));
        } catch (ClassNotFoundException ex) {
            // ignore
        }
        return list;
    }

```

2. 普通平台：标准的JDK ServiceLoader 方式

```

    public static Iterable<ManagedChannelProvider> getCandidatesViaServiceLoader(
        ClassLoader classLoader) {
        return ServiceLoader.load(ManagedChannelProvider.class, classLoader);
    }

```

最重要的**provider()**方法

对于调用者来说，最重要的就是 `provider()` 方法，因为通常调用者都是这样使用：

```

ManagedChannelProvider provider = ManagedChannelProvider.provider();
.....

```

provider()方法的实现很简单，判断一下静态变量 provider，如果为空则抛出异常 ProviderNotFoundException，提示没有可用的 channel service provider：

```
public static ManagedChannelProvider provider() {  
    if (provider == null) {  
        throw new ProviderNotFoundException("No functional channel  
        service provider found. " + "Try adding a dependency on the grp  
        c-okhttp or grpc-netty artifact");  
    }  
    return provider;  
}
```

和装载相关的抽象方法

在load()方法中，每个装载到的候选者，都需要实现这两个方法以便调用。

1. isAvailable()

用来判断这个provider是否可用，需要考虑当前环境。如果返回 false，则其他任何方法都不安全。

```
protected abstract boolean isAvailable();
```

实际在 load() 方法中，所有isAvailable()返回false的候选者都被直接排除。

2. priority()

优先级，每个provider可用的范围是从1到10,需要考虑当前环境。5被视为默认值，然后根据环境检测调整。优先级0 并不是暗示这个provider不工作，只是会排在最后。

```
protected abstract int priority();
```

创建ChannelBuilder的抽象方法

ManagedChannelProvider 的主要功能，还是在于创建适当的 ManagedChannelBuilder，在 ManagedChannelBuilder 中定义了两个抽象方法要求子类做具体实现：

1. builderForAddress()：使用给定的host和端口创建新的builder

```
protected abstract ManagedChannelBuilder< ?> builderForAddress(String name, int port);
```

2. builderForTarget()：使用给定的target URI来创建新的builder

```
protected abstract ManagedChannelBuilder< ?> builderForTarget(String target);
```

类NettyChannelProvider和OkHttpClientProvider

类NettyChannelProvider

用于创建 NettyChannelBuilder 实例的provider。

```
@Internal
public final class NettyChannelProvider extends ManagedChannelProvider {
    @Override
    public boolean isAvailable() {
        // 总是返回true
        return true;
    }

    @Override
    public int priority() {
        // 使用默认优先级 5
        return 5;
    }

    @Override
    public NettyChannelBuilder builderForAddress(String name, int port) {
        // 对接NettyChannelBuilder
        return NettyChannelBuilder.forAddress(name, port);
    }

    @Override
    public NettyChannelBuilder builderForTarget(String target) {
        // 对接NettyChannelBuilder
        return NettyChannelBuilder.forTarget(target);
    }
}
```

类OkHttpChannelProvider

用于创建 OkHttpChannelBuilder 实例的provider。

```
@Internal
public final class OkHttpChannelProvider extends ManagedChannelP
rovider {

    @Override
    public boolean isAvailable() {
        return true;
    }

    @Override
    public int priority() {
        // 会更加当前环境动态调整优先级
        // 如果是 IS_RESTRICTED_APPENGINE 或者 android 环境下，调整为 8,
        此时优先级高于 netty(默认为5)
        // 其他情况下为 3, 此时优先级低于 netty(默认为5)
        return (GrpcUtil.IS_RESTRICTED_APPENGINE || isAndroid()) ? 8
: 3;
    }

    @Override
    public OkHttpChannelBuilder builderForAddress(String name, int
port) {
        return OkHttpChannelBuilder.forAddress(name, port);
    }

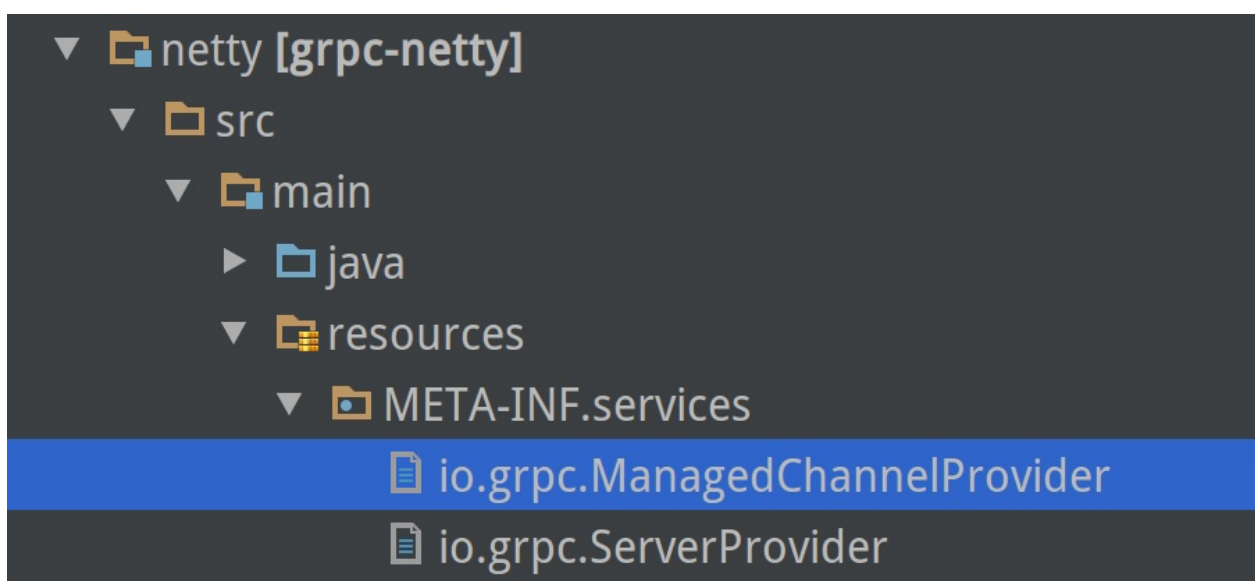
    @Override
    public OkHttpChannelBuilder builderForTarget(String target) {
        return OkHttpChannelBuilder.forTarget(target);
    }
}
```

IS_RESTRICTED_APPENGINE的判断逻辑如下(和 google appengine 有关，细节不追究了)：

```
public static final boolean IS_RESTRICTED_APPENGINE =  
    "Production".equals(System.getProperty("com.google.appengin  
e.runtime.environment"))  
    && "1.7".equals(System.getProperty("java.specification  
.version"));
```

ServiceProvider 的实现

为了实现jdk的ServiceProvider，以便被load()函数装载，okjava 和 netty 在打包的时候都会按照JDK SPI的要求在他们的jar文件中加入SPI的内容，以netty为例：



上图是 grpc-java 中 netty 子项目的相关文件，在resources下的 META-INF.services 目录，存在一个名为 io.grpc.ManagedChannelProvider 文件，其内容为：

```
io.grpc.netty.NettyChannelProvider
```

总结

通过这种标准的SPI的方式，grpc实现了将 channel 的提供者和使用者分离并解藕。

类CallOptions

类CallOptions是新的RPC调用的运行时选项的集合。

类定义

```
package io.grpc;  
@Immutable  
public final class CallOptions {  
}
```

注意@Immutable标签，这个CallOptions是不可变类。

属性和构造函数

CallOptions的代码注释中有讲到：虽然CallOptions是不可变类，但是它的属性并没有声明为final。这样可以在构造函数之外赋值，否则就需要在构造函数中给出长长的一个参数列表。

属性有下面5个，其中有几个声明为容许为null：

```
private Long deadlineNanoTime;  
private Executor executor;  
  
@Nullable  
private String authority;  
  
@Nullable  
private RequestKey requestKey;  
  
@Nullable  
private String compressorName;
```

构造函数比较有意思，第二个版本的构造函数实现了复制功能，相当于创建了一个和给定实例数据完全相同的另一个对象实例：

```
private CallOptions() {  
}  
  
private CallOptions(CallOptions other) {  
    deadlineNanoTime = other.deadlineNanoTime;  
    authority = other.authority;  
    requestKey = other.requestKey;  
    executor = other.executor;  
    compressorName = other.compressorName;  
}
```

为了不使用传统的不可变类的实现方式(属性声明为final，构造函数中一一赋值)，CallOptions中每次赋值都要使用上面的复制构造函数创建一个新的实例对象，然后返回新的实例对象(旧有实例就相当于抛弃了?):

```
public CallOptions withDeadlineNanoTime(@Nullable Long deadlineNanoTime) {  
    CallOptions newOptions = new CallOptions(this);  
    newOptions.deadlineNanoTime = deadlineNanoTime;  
    return newOptions;  
}
```

有个疑惑，按照上面的实现机制，如果CallOptions的使用者要创建一个有多次复制的CallOptions示例，就必须如下编码：

```
CallOptions callOptions = CallOptions.DEFAULT  
    .withDeadlineNanoTime(***)  
    .withAuthority(***)  
    .withCompression(***)  
    .withRequestKey(***)
```

这样实际会创建多次(取决于with()方法的调用次数)实例。不过从grpc的代码实现上看，每次只是在新建一个stub实例时才传入一个 CallOptions 实例，之后的每次调用都将重用这个 CallOptions 实例，所以实际上只是开始时创建多次实例，之后运

行时就不再创建。

属性的详细含义

deadline / 最后期限

`withDeadlineNanoTime()` 返回新的一个CallOptions，最后期限为在给定的绝对值。这个绝对值以纳秒为单位，和每次 `System.nanoTime()` 的时间一致：

```
public CallOptions withDeadlineNanoTime(@Nullable Long deadlineNanoTime) {
    CallOptions newOptions = new CallOptions(this);
    newOptions.deadlineNanoTime = deadlineNanoTime;
    return newOptions;
}
```

`withDeadlineAfter()`方法返回一个最后期限为在给定期限之后的新的CallOptions，可以看到实现的方式就是取当前的 `System.nanoTime()` 然后添加给定的期限：

```
public CallOptions withDeadlineAfter(long duration, TimeUnit unit) {
    return withDeadlineNanoTime(System.nanoTime() + unit.toNanos(duration));
}
```

executor

设置新的executor，用来覆盖 `ManagedChannelBuilder.executor` 指定的默认executor。

requestKey

用于基于亲和度(affinity-based)的路由的request key。

authority / 权限

覆盖 `channel` 声明要连接到的 HTTP/2 的 `authority`。这不是普遍安全的。覆盖容许高级用户为多个服务重用一個单一的 `channel`，甚至这些服务在不同的域名上托管。这将假设服务器是虚拟托管了多个域名并被授权继续这样做。服务提供者极少作出这样的授权。此时，覆盖值没有安全认证，例如保证 `authority` 匹配服务器的 TLS 证书。

compressorName / 压缩器名称

设置为调用使用的压缩方式。压缩器必须是在 `CompressorRegistry` 已知的有效名称。

构建Stub

类DemoServiceBlockingStub

类定义

这个类是通过grpc的proto编译器生成的类，它的package由.proto文件中的java_package 选项指定，如：

```
option java_package = "io.grpc.examples.demo";
```

类定义如下，继承AbstractStub，并实现DemoServiceBlockingClient接口：

```
package io.grpc.examples.demo;

@javax.annotation.Generated("by gRPC proto compiler")
public static class DemoServiceBlockingStub extends io.grpc.stub.
    AbstractStub<DemoServiceBlockingStub>
    implements DemoServiceBlockingClient {
}
```

构造函数

构造函数非常简单，直接调用基类 AbstractStub 的构造函数，传入Channel和CallOptions：

```
private DemoServiceBlockingStub(io.grpc.Channel channel) {
    super(channel);
}

private DemoServiceBlockingStub(io.grpc.Channel channel,
    io.grpc.CallOptions callOptions) {
    super(channel, callOptions);
}
}
```

方法

首先实现了基类 `AbstractStub` 的抽象方法 `build()`，只是简单的构造了自身的一个新的实例：

```
@java.lang.Override
protected DemoServiceBlockingStub build(io.grpc.Channel channel,
    io.grpc.CallOptions callOptions) {
    return new DemoServiceBlockingStub(channel, callOptions);
}
```

业务方法实现

然后就是实现业务定义的 `service` 方法了，这里是定义在 `DemoServiceBlockingClient` 接口中。先看 `DemoServiceBlockingClient` 接口的定义：

```
public static interface DemoServiceBlockingClient {
    public io.grpc.examples.demo.LoginResponse login(io.grpc.examples.demo.LoginRequest request);
    .....
}
```

这些方法对应于 `.proto` 文件中 `service` 定义的 `rpc` 方法：

```
service DemoService {
    rpc login (LoginRequest) returns (LoginResponse) {}
    .....
}
```

`DemoServiceBlockingStub` 中一一实现这些服务方法：

```
@java.lang.Override
public io.grpc.examples.demo.LoginResponse login(io.grpc.examples.demo.LoginRequest request) {
    return blockingUnaryCall(getChannel(), METHOD_LOGIN, getCallOptions(), request);
}
```

类AbstractStub

类AbstractStub是stub实现的通用基类。

类AbstractStub也是生成代码中的stub类的通用基类。这个类容许重定义，例如，添加拦截器到stub。

类定义

```
package io.grpc.stub;  
public abstract class AbstractStub<S extends AbstractStub<S>> {  
}
```

属性和构造函数

类AbstractStub有两个属性：

1. Channel channel
2. CallOptions callOptions

```
private final Channel channel;  
private final CallOptions callOptions;  
  
protected AbstractStub(Channel channel) {  
    this(channel, CallOptions.DEFAULT);  
}  
  
protected AbstractStub(Channel channel, CallOptions callOptions)  
{  
    this.channel = channel;  
    this.callOptions = callOptions;  
}
```

注：类CallOptions 的内容见 [这里](#)

方法

build()抽象方法

定义了抽象方法build()方法来返回一个新的stub，使用给定的Channel和提供的方法配置。

```
protected abstract S build(Channel channel, CallOptions callOptions);
```

- channel：返回的新的stub将使用这个Channel来做通讯。
- callOptions：运行时调用选项，将被应用于这个stub的每次调用。(也就是说这个不可变的callOptions实例之后将在每次调用中重用)

with方法族

定义有多个withxxx()方法，通过创建新的 CallOptions 实例，然后调用上面的 build()方法来返回一个新的stub：

```
public final S withDeadlineNanoTime(@Nullable Long deadlineNanoTime) {  
    return build(channel, callOptions.withDeadlineNanoTime(deadlineNanoTime));  
}  
public final S withDeadlineAfter(long duration, TimeUnit unit) {  
    return build(channel, callOptions.withDeadlineAfter(duration, unit));  
}  
@ExperimentalApi  
public final S withCompression(String compressorName) {  
    return build(channel, callOptions.withCompression(compressorName));  
}
```

也可以替换使用新的Channel，或者在原有的Channel上添加拦截器：

```
public final S withChannel(Channel newChannel) {  
    return build(newChannel, callOptions);  
}  
public final S withInterceptors(ClientInterceptor... interceptors) {  
    return build(ClientInterceptors.intercept(channel, interceptors), callOptions);  
}
```


gRPC客户端源码分析

客户端调用流程

流程概述

标准的grpc client调用代码，最简单的方式，就三行代码：

```
ManagedChannelImpl channel = NettyChannelBuilder.forAddress("127.0.0.1", 6556).build();
DemoServiceGrpc.DemoServiceBlockingStub stub = DemoServiceGrpc.newBlockingStub(channel);
stub.login(LoginRequest.getDefaultInstance());
```

这三行代码，完成了grpc客户端调用服务器端最重要的三个步骤：

1. 创建连接到远程服务器的 channel
2. 构建使用该channel的客户端stub
3. 调用服务方法，执行RPC调用

创建Channel

构建Stub

执行RPC调用

执行RPC调用

gRPC服务器端源码

状态

类Status

注：下面内容翻译自 类Status 的 [javadoc](#)

通过提供标准状态码并结合可选的描述性的信息来定义操作的状态。状态的实例创建是通过以合适的状态码开头并补充额外信息：

```
Status.NOT_FOUND.withDescription("Could not find 'important_file.txt'");
```

对于客户端，每个远程调用在完成时都将返回一个状态。如果发生错误，这个状态将以`RuntimeException`的方式传播到阻塞桩（`blocking stubs`），或者作为一个明确的参数给到监听器。

同样的，服务器可以通过抛出 `StatusRuntimeException` 或者 给回掉传递一个状态来报告状态。

提供工具方法来转换状态到`exception`并从`exception`中解析出状态。

类Status.Code

状态码定义

在`Status`类中，通过内部类 `Status.Code` 以标准java枚举的方式定义了状态码：

```
public final class Status {  
    public enum Code {  
        OK(0),  
        CANCELLED(1),  
        .....  
        UNAUTHENTICATED(16);  
    }  
}
```

这里定了17个（OK和其他16个错误）状态码，它们的详细定义和描述请见 [状态码详细定义](#)。

状态码的属性

Status.Code 有两个属性，类型为整型的value和对应的字符串表示方式的valueAscii：

```
public enum Code {  
    .....  
    private final int value;  
    private final String valueAscii;  
  
    private Code(int value) {  
        this.value = value;  
        this.valueAscii = Integer.toString(value);  
    }  
  
    public int value() {  
        return value;  
    }  
  
    private String valueAscii() {  
        return valueAscii;  
    }  
}
```

Code的status()方法

类Status.Code的status()方法通过使用 STATUS_LIST 来直接返回该状态码对应的Status 对象：

```
private Status status() {  
    return STATUS_LIST.get(value);  
}
```

类Status

STATUS_LIST

静态变量 `STATUS_LIST` 保存了所有的 `Status` 对象，对应每个 `Status.Code` 枚举定义：

```
// Create the canonical list of Status instances indexed by their code values.
private static final List<Status> STATUS_LIST = buildStatusList();

private static List<Status> buildStatusList() {
    TreeMap<Integer, Status> canonicalizer = new TreeMap<Integer, Status>();
    // 将所有的Code的枚举定义都遍历
    for (Code code : Code.values()) {
        Status replaced = canonicalizer.put(code.value(), new Status(code));
        if (replaced != null) {
            //去重处理，有些奇怪这里可能重复吗？输入的可是枚举定义，value按说不会定义错
            throw new IllegalStateException("Code value duplication between "
                + replaced.getCode().name() + " & " + code.name());
        }
    }
    return Collections.unmodifiableList(new ArrayList<Status>(canonicalizer.values()));
}
```

直接用 `value` 做下标，因此枚举定义的 `value` 就是从0开始。

Status的静态实例

`Status` 中为每个 `Status.Code` 定义了一对一的静态的 `Status` 实例：

```
public static final Status OK = Code.OK.status();
public static final Status CANCELLED = Code.CANCELLED.status();
.....
public static final Status DATA_LOSS = Code.DATA_LOSS.status();
```

注意 `Code.status()` 方法是调 `STATUS_LIST.get(value)` 来获取 `Status` 实例。

Status的属性 and 构造函数

`Status`的属性，注意都是`final`不可变：

```
// code 用来保存状态码
private final Code code;
// description 是状态描述信息
private final String description;
// cause 是关联的异常
private final Throwable cause;

private Status(Code code) {
    // description 和 cause 都可以为null
    this(code, null, null);
}

private Status(Code code, @Nullable String description, @Nullable
    e Throwable cause) {
    this.code = checkNotNull(code);
    this.description = description;
    this.cause = cause;
}
```

注意`Status`是注明 `@Immutable` 的，而且这个类是 `final`，不可以继承：

```
@Immutable
public final class Status {}
```

构造 Status 对象的静态方法

1. 静态方法`fromCodeValue()`根据给定的状态码构建`Status`对象：


```
public static Status fromCodeValue(int codeValue) {
    if (codeValue < 0 || codeValue > STATUS_LIST.size()) {
        return UNKNOWN.withDescription("Unknown code " + codeValue);
    } else {
        return STATUS_LIST.get(codeValue);
    }
}
```

对于有效的 code 值(0到 STATUS_LIST.size()), 直接返回对应的保存在 STATUS_LIST 中的实例, 这样得到的实例和前面的静态定义实际是同一个实例。

注意: 返回的 Status 实例中 description 和 cause 属性都是null。

2. 静态方法 fromCode() 根据给定的Code对象构建Status对象:

```
public static Status fromCode(Code code) {
    return code.toStatus();
}
```

//而toStatus()的实现非常简单, 直接从 STATUS_LIST 里面取值。

```
public enum Code {
    public Status toStatus() {
        return STATUS_LIST.get(value);
    }
}
```

3. 静态方法 fromThrowable() 根据给定的 Throwable 对象构建Status对象:

```
public static Status fromThrowable(Throwable t) {
    Throwable cause = checkNotNull(t);
    // 循环渐进，逐层检查
    while (cause != null) {
        if (cause instanceof StatusException) {
            //StatusException就直接取status属性
            return ((StatusException) cause).getStatus();
        } else if (cause instanceof StatusRuntimeException) {
            //StatusRuntimeException也是直接取status属性
            return ((StatusRuntimeException) cause).getStatus();
        }
        //不是的话就继续检查cause
        cause = cause.getCause();
    }

    //最后如果还是找不到任何Status，就只能给 UNKNOWN
    return UNKNOWN.withCause(t);
}
```

修改 Status 对象的属性

由于 Status 对象是不可变的，因此如果需要修改属性，只能重新构建一个新的实例。

1. code 属性通常不需要修改
2. 设置 cause 属性

```
// 使用给定 cause 创建派生的 Status 实例。
// 但是不管如何，cause 不会从服务器传递到客户端
public Status withCause(Throwable cause) {
    if (Objects.equal(this.cause, cause)) {
        return this;
    }
    return new Status(this.code, this.description, cause);
}
```

3. 设置 description 属性

```
// 使用给定 description 创建派生的 Status 实例。
public Status withDescription(String description) {
    if (Objects.equal(this.description, description)) {
        return this;
    }
    return new Status(this.code, description, this.cause);
}
```

也可以在现有的 description 属性基础上增加额外的信息：

```
// 在现有 description 的基础上增加额外细节来创建派生的 Status 实例。

public Status augmentDescription(String additionalDetail) {
    if (additionalDetail == null) {
        return this;
    } else if (this.description == null) {
        // 如果原来 description 为null，直接设置
        return new Status(this.code, additionalDetail, this.cause);
    } else {
        // 如果原来 description 不为null，通过"\n"连接起来
        return new Status(this.code, this.description + "\n"
+ additionalDetail, this.cause);
    }
}
```

Status 的方法

1. `isOk()` 简单判断是否OK

```
public boolean isOk() {
    return Code.OK == code;
}
```

2. `asRuntimeException()` 方法将当前 Status 对象转为一个 RuntimeException

```
public StatusRuntimeException asRuntimeException() {  
    return new StatusRuntimeException(this);  
}
```

注意这里得到的 `StatusRuntimeException` 对象携带了当前 `Status`，可以通过方法 `fromThrowable()` 找回这个 `Status` 实例。

3. 带跟踪元数据的 `asRuntimeException()` 方法将当前 `Status` 对象转为一个 `RuntimeException` 并携带跟踪元数据

```
public StatusRuntimeException asRuntimeException(Metadata trailers) {  
    return new StatusRuntimeException(this, trailers);  
}
```

4. `asException()` 功能类似，但是得到的是 `Exception`

状态码详细定义

注：在 `Status.Code` 中通过枚举的方式定义状态码，这些状态码的定义非常的有参考和借鉴价值，因此详细翻译了一遍。

定义和注释

- **OK(0)：成功**

操作成功完成

- **CANCELLED(1)：被取消**

操作被取消（通常是被调用者取消）

- **UNKNOWN(2)：未知**

未知错误。这个错误可能被返回的一个例子是，如果从其他地址空间接收到的状态值属于在当前地址空间不知道的错误的空间（注：看不懂。。。）。此外，API发起的没有返回足够信息的错误也可能被转换到这个错误。

- **INVALID_ARGUMENT(3)：无效参数**

客户端给出了一个无效参数。注意，这和 `FAILED_PRECONDITION` 不同。`INVALID_ARGUMENT` 指明是参数有问题，和系统的状态无关。

- **DEADLINE_EXCEEDED(4)：超过最后期限**

在操作完成前超过最后期限。对于修改系统状态的操作，甚至在操作被成功完成时也可能返回这个错误。例如，从服务器返回的成功的应答可能被延迟足够长时间以至于超过最后期限。

- **NOT_FOUND(5)：无法找到**

某些请求实体(例如文件或者目录)无法找到

- **ALREADY_EXISTS(6)：已经存在**

某些我们试图创建的实体(例如文件或者目录)已经存在

- PERMISSION_DENIED(7): 权限不足

调用者没有权限来执行指定操作。PERMISSION_DENIED 不可以用于因为某些资源被耗尽而导致的拒绝（对于这些错误请使用 RESOURCE_EXHAUSTED）。当调用者无法识别身份时不要使用 PERMISSION_DENIED（对于这些错误请使用 UNAUTHENTICATED）

- RESOURCE_EXHAUSTED(8): 资源耗尽

某些资源已经被耗尽，可能是用户配额，或者可能是整个文件系统没有空间。

- FAILED_PRECONDITION(9): 前置条件失败

操作被拒绝，因为系统不在这个操作执行所要求的状态下。例如，要被删除的目录是非空的，rmdir操作用于非目录等。

下面很容易见分晓的测试可以帮助服务实现者来决定使用 FAILED_PRECONDITION, ABORTED 和 UNAVAILABLE:

1. 如果客户端可以重试刚刚这个失败的调用，使用 UNAVAILABLE
2. 如果客户端应该在更高级别做重试（例如，重新开始一个读-修改-写序列操作），使用 ABORTED
3. 如果客户端不应该重试，直到系统状态被明确修复，使用 FAILED_PRECONDITION。例如，如果 "rmdir" 因为目录非空而失败，应该返回 FAILED_PRECONDITION，因为客户端不应该重试，除非先通过删除文件来修复目录。

- ABORTED(10): 中途失败

操作中途失败，通常是因为并发问题如时序器检查失败，事务失败等。

- OUT_OF_RANGE(11): 超出范围

操作试图超出有效范围，例如，搜索或者读取超过文件结尾。

和 INVALID_ARGUMENT 不同，这个错误指出的问题可能被修复，如果系统状态修改。例如，32位文件系统如果被要求读取不在范围 $[0, 2^{32}-1]$ 之内的 offset 将生成 INVALID_ARGUMENT，但是如果被要求读取超过当前文件大小的 offset 时将生成 OUT_OF_RANGE。

在 FAILED_PRECONDITION 和 OUT_OF_RANGE 之间有一点重叠。当 OUT_OF_RANGE 适用时我们推荐使用 OUT_OF_RANGE（更具体的错误）。

- UNIMPLEMENTED(12): 未实现

操作没有实现，或者在当前服务中没有支持/开启。

- INTERNAL(13)：内部错误

内部错误。意味着某些底层系统期待的不变性被打破。如果看到这些错误，说明某些东西被严重破坏。

- UNAVAILABLE(14)：不可用

服务当前不可用。这大多数可能是一个临时情况，可能通过稍后的延迟重试而被纠正。

- DATA_LOSS(15)：数据丢失

无法恢复的数据丢失或者损坏。

- UNAUTHENTICATED(16)：未经认证

请求没有操作要求的有效的认证凭证。

思考

有个疑问：上述状态码是通过枚举的方式定义的，如果需要做扩展（应对应用自身的特殊的状态），该如何进行？

类StatusException

gRPC中定义了两个和 Status 相关的异常，分别是 StatusException 和 StatusRuntimeException。以异常的实行来表示并传递状态信息。

类StatusException

代码足够简单，继承自Exception，通过传递一个 Status 对象实例来构建，并从 Status对象实例中获取异常信息。

```
public class StatusException extends Exception {
    private static final long serialVersionUID = -660954903976144640L;
    private final Status status;
    private final Metadata trailers;

    public StatusException(Status status) {
        this(status, null);
    }

    @ExperimentalApi
    // 使用状态和跟踪元数据来构建 exception
    // 这个方法是 1.0.0-pre2 之后才增加的
    public StatusException(Status status, @Nullable Metadata trailers) {
        super(Status.formatThrowableMessage(status), status.getCause());
        this.status = status;
        this.trailers = trailers;
    }

    public final Status getStatus() {
        return status;
    }
}
```


Status.formatThrowableMessage()方法用于从 Status 中获取异常的 message 信息：

1. 异常的message

通过 Status 的formatThrowableMessage()方法从Status中得到message。

formatThrowableMessage()方法的具体实现代码：

```
static String formatThrowableMessage(Status status) {  
    if (status.description == null) {  
        // 如果没有description，则直接取code.toString()  
        // code是一个枚举，code.toString()方法里面取的是枚举的name属性  
        // 也就是 INVALID_ARGUMENT 之类的字符串  
        return status.code.toString();  
    } else {  
        // 如果有description，则将code(同上实际是枚举的name)和description拼起来  
        return status.code + ": " + status.description;  
    }  
}
```

2. 异常的casue：直接取Status对象的casue

类StatusRuntimeException

代码和实现与StatusException完全一致，只是继承的是RuntimeException。

异常处理的流程分析

前言

在 gRPC 的新版本(1.0.0-pre2)中，为了方便传递 debug 信息，在 `StatusException` 和 `StatusRuntimeException` 中增加了名为 `Trailer` 的 `Metadata`。

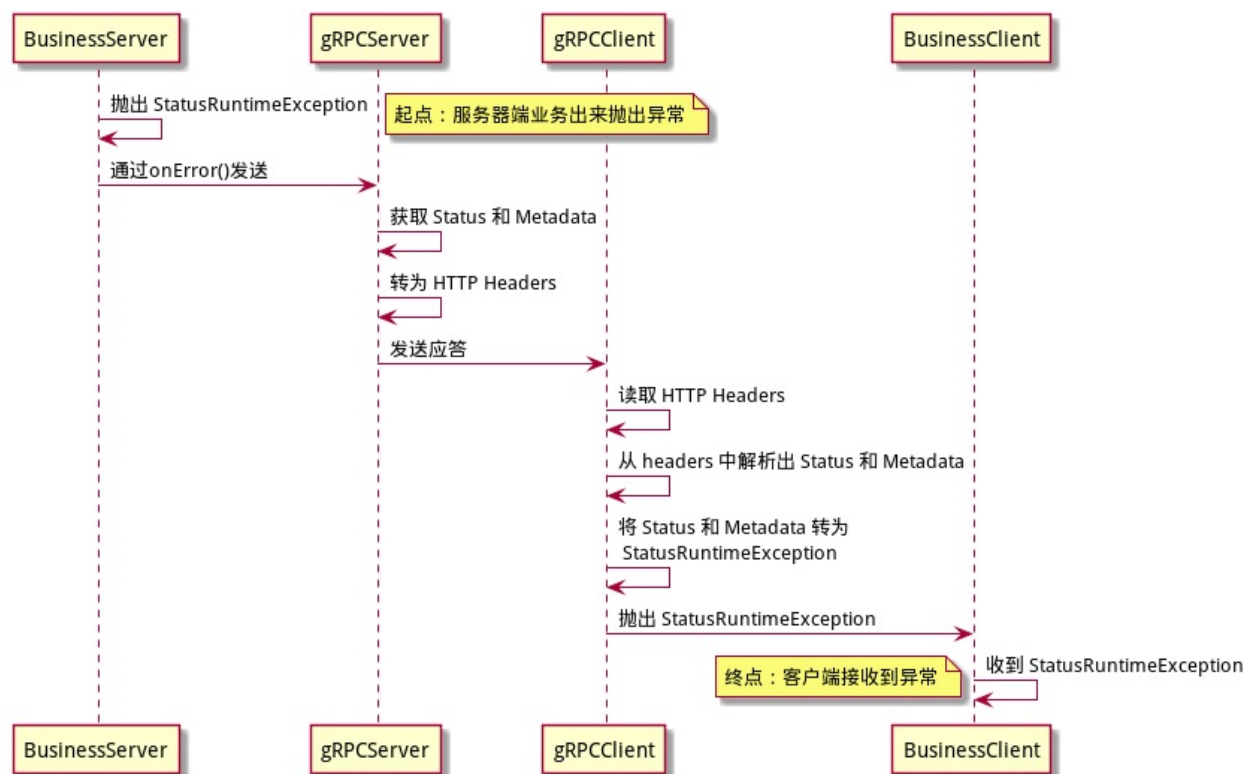
注：在此之前，`Status`(和 `Status` 映射的 `StatusException`)只有两个字段可以传递信息：1. `status code` 2. `status description` 如果需要传递其他信息则无计可施。在引入 `Metadata` 之后，终于可以通过使用 `Metadata` (映射到 HTTP2 header) 来传递更多信息。

至此，gRPC 中处理异常的设计已经基本成型，其核心设计就是通过 `Status` 和 `Metadata` 来传递异常信息，服务器端抛出异常到客户端的过程，实际是对异常和 `Status/Metadata` 的传递和转换的过程：

1. 服务器端抛出异常
2. 异常转为 `Status/Metadata`
3. `Status/Metadata` 传递到客户端
4. 客户端将 `Status/Metadata` 转回异常

而 `Status/Metadata` 在服务器端和客户端的传输中，是通过 HTTP header 来实现的。

整个异常处理的流程概述如图：



最终实现的目标：当服务器端抛出异常时，客户端就会抛出对应的异常。

下面我们对整个流程做详细的代码分析。

Metadata 的准备工作

异常中引入 Metadata

在 StatusRuntimeException 和 StatusException，增加了一个新的构造函数，可以传入 Metadata 并保存起来，同时提供一个 getter 方法用来获取保存的 Metadata：

```
private final Metadata trailers;

public StatusRuntimeException(Status status, @Nullable Metadata trailers) {
    super(Status.formatThrowableMessage(status), status.getCause());
    this.status = status;
    this.trailers = trailers;
}

public final Metadata getTrailers() {
    return trailers;
}
```

Status 中引入 Metadata

看 class Status 的相关代码实现，这里主要是做好在 Status 到 异常的相互转换过程中处理 Metadata 的准备：

1. 定义了两个 Metadata.Key，分别用于 status 的状态码和消息：

```
/**
 * 用于绑定状态码到 trailing metadata 的key.
 */
public static final Metadata.Key<Status> CODE_KEY
    = Metadata.Key.of("grpc-status", new StatusCodeMarshaller());

/**
 * 绑定状态消息到 trailing metadata 的key.
 */
@Internal
public static final Metadata.Key<String> MESSAGE_KEY
    = Metadata.Key.of("grpc-message", STATUS_MESSAGE_MARSHALLER);
```

2. 从异常(包括cause 链)的中提取出 error trailers

```
public static Metadata trailersFromThrowable(Throwable t) {
    Throwable cause = checkNotNull(t, "t");
    while (cause != null) {
        if (cause instanceof StatusException) {
            return ((StatusException) cause).getTrailers();
        } else if (cause instanceof StatusRuntimeException) {
            return ((StatusRuntimeException) cause).getTrailers();
        }
        cause = cause.getCause();
    }
    return null;
}
```

3. 重载两个方法，在从 Status 到 异常 的转变过程中容许加入 Metadata 信息

```
public StatusRuntimeException asRuntimeException(Metadata trailers) {
    return new StatusRuntimeException(this, trailers);
}
public StatusException asException(Metadata trailers) {
    return new StatusException(this, trailers);
}
```

服务器端流程

以onError()方式发送异常

当服务器端需要抛出异常，尤其是 StatusRuntimeException 和 StatusException 时，正确的姿势是通过 StreamObserver 对象的 onError() 方法发送异常信息到客户端：

```
public void sayHello(SayHelloRequest request, StreamObserver<SayHelloResponse> responseObserver) {
    responseObserver.onError(new StatusRuntimeException(Status.ALREADY_EXISTS));
}
```

onError()方法在 ServerCalls.ServerCallStreamObserverImpl 中的实现：

```
public void onError(Throwable t) {
    // 从 Throwable 中获取 Metadata
    Metadata metadata = Status.trailersFromThrowable(t);
    if (metadata == null) {
        // 如果没有找到，即异常中没有设置 Metadata，则只能 new 一个空的 Metadata
        metadata = new Metadata();
    }
    // 将 异常 转为 Status 对象，然后加上 Metadata ，一起发送给客户端
    call.close(Status.fromThrowable(t), metadata);
}
```

call 的 close() 方法在 ServerCallImpl 中的实现：

```
public void close(Status status, Metadata trailers) {
    checkState(!closeCalled, "call already closed");
    closeCalled = true;
    // 发送给 stream
    stream.close(status, trailers);
}
```

stream 的 close() 方法在 AbstractServerStream 中的实现：

```
public final void close(Status status, Metadata trailers) {
    Preconditions.checkNotNull(status, "status");
    Preconditions.checkNotNull(trailers, "trailers");
    if (!outboundClosed) {
        outboundClosed = true;
        endOfMessages();
        //将 Status 添加到 Metadata
        addStatusToTrailers(trailers, status);
        //将 Metadata 发送出去，注意此时 Status 是没有发送的，只有 Metad
ata
        abstractServerStreamSink().writeTrailers(trailers, headers
Sent);
    }
}

private void addStatusToTrailers(Metadata trailers, Status statu
s) {
    // 安全起见，删除 Metadata 中可能存在的 CODE_KEY 和 MESSAGE_KEY
    trailers.removeAll(Status.CODE_KEY);
    trailers.removeAll(Status.MESSAGE_KEY);
    // 将 status 放置到 Metadata
    trailers.put(Status.CODE_KEY, status);
    if (status.getDescription() != null) {
        // 将 description 放置到 Metadata
        trailers.put(Status.MESSAGE_KEY, status.getDescription());
    }
}
```

writeTrailers() 方法在 NettyServerStream 中的实现：

```
public void writeTrailers(Metadata trailers, boolean headersSent)
{
    Http2Headers http2Trailers = Utils.convertTrailers(trailers, h
headersSent);
    writeQueue.enqueue(
        new SendResponseHeadersCommand(transportState(), http2Trai
lers, true), true);
}
```

总结：服务器端抛出异常之后，异常信息被转化为 **Status** 和 **Metadata**，然后 **Status** 也最终被转为 **Metadata**，最后以 **HTTP Header** 的方式发送给客户端。

直接抛出异常

如果 gRPC 服务器端不是以标准的 `onError()` 方法发送异常，而是以直接抛出异常的方式结束处理流程，则此时的处理方式有所不同：**Metadata** 信息会被忽略，而不是传递给客户端！

注：谨记标准的符合 gRPC 要求的方式是通过 `onError()` 方法，而不是直接抛出异常。

当服务器端抛出异常时，处理这个异常的代码在 `ServerImpl.JumpToApplicationThreadServerStreamListener.halfClosed()` 中：


```

private static class JumpToApplicationThreadServerStreamListener
implements ServerStreamListener {
    .....
    public void halfClosed() {
        callExecutor.execute(new ContextRunnable(context) {
            @Override
            public void runInContext() {
                try {
                    getListener().halfClosed(); //服务器端抛出的异常会在这里
跑出来
                } catch (RuntimeException e) {
                    // 这里没有从异常中获取 Metadata，而是 new 了一个新的空的
                    internalClose(Status.fromThrowable(e), new Metadata(
));
                    throw e;
                } catch (Throwable t) {
                    // 同上
                    internalClose(Status.fromThrowable(t), new Metadata(
));
                    throw new RuntimeException(t);
                }
            }
        });
    }
}

```

可以看到此时之前异常带的 **metadata** 信息是会被一个空的对象替代。

注：关于这个事情，我开始认为是一个实现bug，因此提交了一个 Pull Request 给 gRPC，但是后来 gRPC 的开发人员解释说这个是故意设计的，为的就是要强制服务器端使用 `onError()` 方法而不是直接抛出异常。详细情况请见 [metadata is lost when server sends StatusRuntimeException](#)

解决方案

虽然 gRPC 官方推荐用 `onError()` 处理异常，但是实际上在实践时需要每个业务方法都要来一个大的 `try catch`。这使得代码冗余而烦琐。

解决的方式，是自己写一个 `ServerInterceptor`，实现一个

`io.grpc.ServerCall.Listener` 来统一处理

```
class ExceptionInterceptor implements ServerInterceptor {
    public <ReqT, RespT> ServerCall.Listener<ReqT> interceptCall
    (
        ServerCall<ReqT, RespT> call, Metadata headers,
        ServerCallHandler<ReqT, RespT> next) {
        ServerCall.Listener<ReqT> reqTListener = next.startCall(
call, headers);
        return new ExceptionListener(reqTListener, call);
    }
}

class ExceptionListener extends ServerCall.Listener {
    .....
    public void onHalfClose() {
        try {
            this.delegate.onHalfClose();
        } catch (Exception t) {
            // 统一处理异常
            ExtendedStatusRuntimeException exception = fromThrow
able(t);
            // 调用 call.close() 发送 Status 和 metadata
            // 这个方式和 onError()本质是一样的
            call.close(exception.getStatus(), exception.getTrail
ers());
        }
    }
}
```

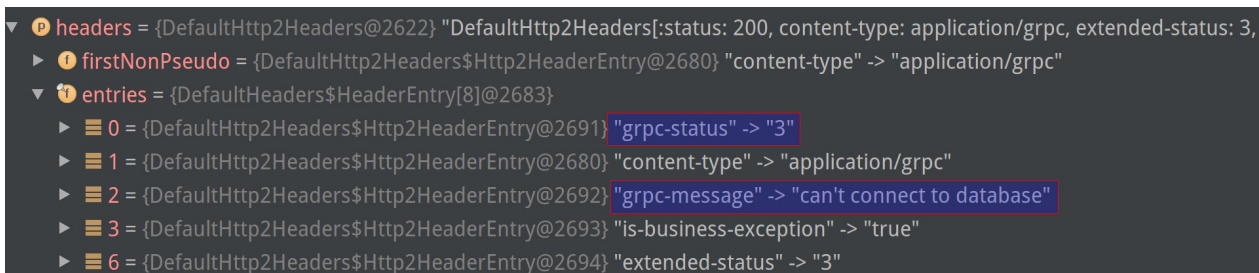
客户端流程

接收应答读取Header

客户端在发送请求收到应答之后，在 `DefaultHttp2FrameReader` 中读取 frame，前面讲过异常会以 HTTP header 的方式发送过来，在客户端反应为客户端收到 Headers Frame：

```
private void processPayloadState(ChannelHandlerContext ctx, Byte
Buf in, Http2FrameListener listener){
    switch (frameType) {
        .....
        case HEADERS:
            readHeadersFrame(ctx, payload, listener);
            break;
    }
}
```

读取出来的应答和 header 内容如下图：



```
▼ headers = {DefaultHttp2Headers@2622} "DefaultHttp2Headers[:status: 200, content-type: application/grpc, extended-status: 3,
  ► firstNonPseudo = {DefaultHttp2Headers$Http2HeaderEntry@2680} "content-type" -> "application/grpc"
  ▼ entries = {DefaultHttp2Headers$HeaderEntry@2683}
    ► 0 = {DefaultHttp2Headers$Http2HeaderEntry@2691} "grpc-status" -> "3"
    ► 1 = {DefaultHttp2Headers$Http2HeaderEntry@2680} "content-type" -> "application/grpc"
    ► 2 = {DefaultHttp2Headers$Http2HeaderEntry@2692} "grpc-message" -> "can't connect to database"
    ► 3 = {DefaultHttp2Headers$Http2HeaderEntry@2693} "is-business-exception" -> "true"
    ► 6 = {DefaultHttp2Headers$Http2HeaderEntry@2694} "extended-status" -> "3"
```

这里可以看到以下内容：

1. HTTP 应答的状态码是200,表示成功，即使 gRPC 服务器端返回异常表示业务处理失败。因此，用 HTTP 状态码来评估服务器是否处理正常是没有意义的。
2. HTTP 应答的 content-type 是 "application/grpc"
3. grpc-status 和 grpc-message 两个 header 对应 Status 对象的 code 和 description
4. 其他的 header 对应 Metadata 中的数据，比如上面的 extended-status 和 is-business-exception

转换为 status 和 trailer

之后在 Http2ClientStream 中将 header 转成 trailer (也就是Metadata)：

```
void transportHeadersReceived(Http2Headers headers, boolean endOf
Stream) {
    if (endOfStream) {
        transportTrailersReceived(Utils.convertTrailers(headers));
    }
    .....
}
```

然后从 trailer 中获取数据转成 Status 对象：

```
protected void transportTrailersReceived(Metadata trailers) {
    .....
    // 从 metadata 中获取 Status
    Status status = statusFromTrailers(trailers);
    // 清理不再需要的 header
    stripTransportDetails(trailers);
    // 继续处理
    inboundTrailersReceived(trailers, status);
}

private Status statusFromTrailers(Metadata trailers) {
    // 从 metadata 中的两个 key 获取 Status，注意 message 有可能为空
    Status status = trailers.get(Status.CODE_KEY);
    String message = trailers.get(Status.MESSAGE_KEY);
    if (message != null) {
        status = status.augmentDescription(message);
    }
    return status;
}

private static void stripTransportDetails(Metadata metadata) {
    // 去除传输细节
    // 实际就是删除 http 状态码的 header 和 Status 的两个属性的 header
    // 清理之后剩下的就是服务器端传过来的 metadata
    metadata.removeAll(HTTP2_STATUS);
    metadata.removeAll(Status.CODE_KEY);
    metadata.removeAll(Status.MESSAGE_KEY);
}
```

此时 Status 和 Metadata 已经从 header 中还原出来，和服务器端发送的保持一致，后面就是传递和处理过程。

传递 Status 和 Metadata

Status 和 Metadata 被一路传递，在 DelayedStreamListener.closed()方法，调用一个任务，在 run()方法中调用 Listener 的 closed()方法：

```
@Override
public void closed(final Status status, final Metadata trailers)
{
    delayOrExecute(new Runnable() {
        @Override
        public void run() {
            realListener.closed(status, trailers);
        }
    });
}
```

ClientCallImpl.ClientStreamListenerImpl，在closed()方法中创建了一个StreamClosed的任务，扔给callExecutor：

```
public void closed(Status status, Metadata trailers) {
    .....
    final Status savedStatus = status;
    final Metadata savedTrailers = trailers;
    class StreamClosed extends ContextRunnable {
        StreamClosed() {
            super(context);
        }

        @Override
        public final void runInContext() {
            if (closed) {
                // We intentionally don't keep the status or metadata
                a from the server.
                return;
            }
            close(savedStatus, savedTrailers);
        }
    }

    callExecutor.execute(new StreamClosed());
}
```

这个任务在执行时调用 `ClientCallImpl` 的 `close()` 方法，然后调用 `observer.onClose()` 方法：

```
private void close(Status status, Metadata trailers) {
    closed = true;
    cancelListenersShouldBeRemoved = true;
    try {
        observer.onClose(status, trailers);
    } finally {
        removeContextListenerAndCancelDeadlineFuture();
    }
}
```

将 **Status** 和 **Metadata** 转换为异常

最后进入 `ClientCalls.UnaryStreamToFuture` 的 `onClose()` 方法，这里做最终的 `Status` 和异常 的转换：

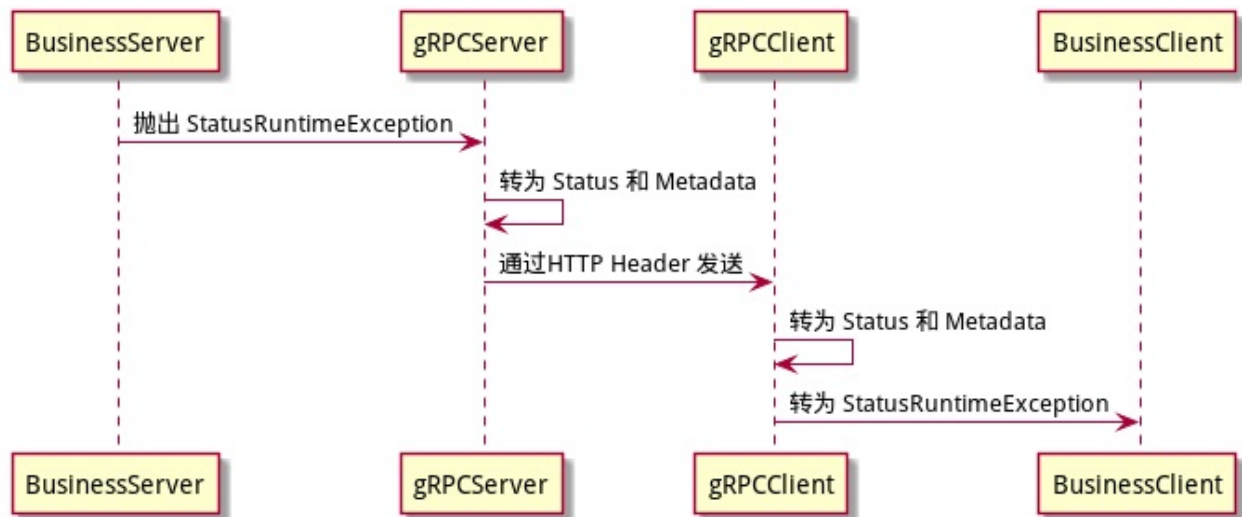
```
private static class UnaryStreamToFuture<RespT> extends ClientCa
ll.Listener<RespT> {
    .....
    public void onClose(Status status, Metadata trailers) {
        if (status.isOk()) {
            .....
        } else {
            // 当 Status 不是OK时，将 Status 和 Metadata 转为异常
            // 然后交给 responseFuture，后面客户端调用就会通过 responseFu
ture 得到这个异常
            responseFuture.setException(status.asRuntimeException(tr
ailers));
        }
    }
    .....
}
```

总结

整个异常处理的流程，总结起来就是两点：

1. 异常的传输是通过 Status/Metadata 来实现
2. Status/Metadata 的传输是通过 HTTP Header 来实现的

简化之后的流程图：



实战

集成Spring Boot

Spring Boot是个人非常喜欢的一个微服务框架，因此很希望能集成gRPC和spring boot.

下面是在网上找到的一点资料。

相关资料

搜索了一下，找到一些资料：

- [Using Spring Boot together with gRPC and Protobuf](#)
- [Using Google Protocol Buffers with Spring MVC-based REST Services](#)

相关项目

后来发现一个有意义的新项目：

- [grpc-spring-boot-starter](#)

注：上面这个项目中集成springboot和grpc的方式非常棒，我们后面参考了这种做法。鸣谢原作者！

文档生成

protoc-gen-doc

<https://github.com/estan/protoc-gen-doc>

这是一个Google Protocol Buffers编译器(protoc)的文档生成插件。这个插件可以从.proto文件中的注释内容生成HTML, DocBook 或者 Markdown 文档。

安装

参考 [protoc-gen-doc Installation](#) 章节的信息。

linux安装

对于ubuntu系统，参考 [protoc-gen-doc](#) 的说明。

对于 xUbuntu 14.04，请运行以下命令：

```
sudo sh -c "echo 'deb http://download.opensuse.org/repositories/home:/estan:/protoc-gen-doc/xUbuntu_14.04/ /' >> /etc/apt/sources.list.d/protoc-gen-doc.list"
sudo apt-get update
sudo apt-get install protoc-gen-doc
```

windows

protoc-gen-doc的github项目的 [release page](#)上，可以找到windows的版本的下载。

执行

```
cd dolphin-demo/contract
protoc --doc_out=html,index.html:target/contract-doc src/main/proto/*.proto
```

可惜，报错：

```
src/main/proto/user.proto:1:10: Unrecognized syntax identifier "
proto3". This parser only recognizes "proto2".
```

目前版本还不支持proto3！真是遗憾。

2016-08-07 更新：发现上面的错误提示只是 protoc 版本的问题，只要升级 protoc 版本到v3.0.0，这个 proto-gen-doc 插件依然可以正常工作！详细做法请见下一节 "支持proto3".

支持proto3

安装 protoc-gen-doc

简单遵循安装要求即可：

<https://github.com/estan/protoc-gen-doc#installation>

安装完成之后的protoc是2.5.0版本，无法处理proto3的文件。因此我们需要升级替换protoc为v3.0.0版本。

升级protoc

使用预编译版本

1. 下载

请先在 [protobuf 的发布页面](#) 中找到对应版本的 download，然后下载对应版本的 protoc, 如 v3.0.0的 linux 64位版本：

- [protoc-3.0.0-linux-x86_64.zip](#)

2. 清理

如果之前有安装过 protoc 的老版本，如2.5.0版本，则需要在安装前删除旧有版本的protoc和include文件，操作如下：

```
$ protoc --version
libprotoc 2.5.0
# 上面老版本是 2.5.0，需要删除
which protoc
/usr/bin/protoc
# 删除 protoc
sudo rm /usr/bin/protoc
# 删除默认的include文件
sudo rm -rf /usr/include/google/protobuf/
```

3. 安装

解压缩得到的zip文件，然后执行下面的操作，复制protoc和include文件：

```
mkdir temp
mv protoc-3.0.0-linux-x86_64.zip temp
cd temp
unzip protoc-3.0.0-linux-x86_64.zip
```

输出如下(直接解压缩到当前目录，所以解压前最好准备一个临时目录，如上面shell命令中的temp目录):

```
Archive:  protoc-3.0.0-linux-x86_64.zip
creating: include/
creating: include/google/
creating: include/google/protobuf/
inflating: include/google/protobuf/struct.proto
inflating: include/google/protobuf/type.proto
inflating: include/google/protobuf/descriptor.proto
inflating: include/google/protobuf/api.proto
inflating: include/google/protobuf/empty.proto
creating: include/google/protobuf/compiler/
inflating: include/google/protobuf/compiler/plugin.proto
inflating: include/google/protobuf/any.proto
inflating: include/google/protobuf/field_mask.proto
inflating: include/google/protobuf/wrappers.proto
inflating: include/google/protobuf/timestamp.proto
inflating: include/google/protobuf/duration.proto
inflating: include/google/protobuf/source_context.proto
creating: bin/
inflating: bin/protoc
inflating: readme.txt
```

开始复制文件并修改文件权限为755：

```
# 复制protoc文件
sudo cp bin/protoc /usr/bin/
sudo chmod 755 /usr/bin/protoc
# 复制include文件
sudo cp -r include/google/protobuf/ /usr/include/google/
sudo chmod -R 755 /usr/include/google/protobuf
```

生成文档

参考 protoc-gen-doc 的使用说明：

<https://github.com/estan/protoc-gen-doc#invoking-the-plugin>

注意：存放生成文件的目录必须预先建立，否则会报错。

```
// 我们的proto文件在这里
cd contract/src/main/proto
// 准备生成HTML文件，并放置在target/contract-doc目录下
mkdir ../../../../target/contract-doc
protoc --doc_out=html,index.html:../../../../target/contract-doc userService.proto
```

解决 import 问题

执行 protoc 时，如果 userService.proto 还 import 了 google/protobuf 之外的其他的 .proto 文件，则会因为无法找到需要的 import 文件而报错。

有三种方式解决：

1. 直接将要import的proto文件复制到当前目录，保证路径正确
2. 将需要import的proto文件复制到 /usr/include/ 目录

```
sudo cp -r dolphin/ /usr/include/
sudo chmod -R 755 /usr/include/dolphin/
```

3. 使用 --proto_path=PATH 在 protoc 的命令行参数中添加import的路径，而且这个选项可以使用多次

总结

现在至少可以用了，可以得到生成的HTML文件，虽然还有很多不足，后续再更新。

后续工作

后面就可以考虑：

1. 在jenkins上建立上述环境
2. 然后添加job来为各个项目的contract文件生成HTML文档
3. 再将生成的文档统一发布到nginx之类的服务器下

这样就可以实现一个简单的文档中心，用于各个服务API文档的统一维护和发布。

build 插件

原有插件生成的 HTML 文件内容和格式并不理想，考虑自行调整。

因此 fork 了原有仓库，准备动手修改。

这样就有必要能自己从c的源代码开始编译打包。

参考原有的插件打包说明：

<https://github.com/skyao/protoc-gen-doc/blob/master/BUILDING.md>

准备工作

按照要求，需要准备两个东西：

- Protocol Buffers library from Google
- QtCore from Qt 5

先执行命令：

```
sudo apt-get install qt5-qmake qt5-default libprotobuf-dev proto  
buf-compiler libprotoc-dev
```

另外g++肯定是必备的了，没有安装的话先安装：

```
sudo apt-get install g++
```

make

在插件的代码根目录下，执行下面命令：

```
qmake  
make
```

此时如果发现报错，无法include：


```
/usr/include/google/protobuf/compiler/plugin.h:58:42: fatal error: google/protobuf/stubs/common.h: No such file or directory
#include <google/protobuf/stubs/common.h>
                                         ^
compilation terminated.
make: *** [main.o] Error 1
```

说明第一个前提条件 "Protocol Buffers library from Google" 没有满足，需要先安装。

安装 Protocol Buffers library

参考 protocol buffer 的 `C++ Installation - Unix`：

<https://github.com/google/protobuf/blob/master/src/README.md#c-installation---unix>

先安装各种工具：

```
sudo apt-get install autoconf automake libtool curl make g++ unzip
```

然后准备开始build。注意这里有两个方式，对应着protocol release页面有两个下载地址：

- protobuf-cpp-3.0.0.zip
- Source code (tar.gz)
- 从source code开始：下载 `Source code (tar.gz)`，然后开始build，因为这个包中没有configure脚本文件，因此需要执行 `./autogen.sh` 来生成。而在这个过程中需要下载jmock来做测试，但是jmock下载时又遇到无法下载的问题。所以不建议用这种方式
- 从release package中，也就是下载 `protobuf-cpp-3.0.0.zip`，这里面有现成的configure脚本文件，一次执行下面的命令：

```
./configure
make
make check
sudo make install
sudo ldconfig
```

make完成之后，上面缺失的头文件就可以在下列地址找到了：

```
/usr/local/include/google/protobuf/stubs/common.h
```

清理

用上面的方式安装 protocol buffer 之后，protoc被安装在新的位置了：

```
$which protoc
/usr/local/bin/protoc
```

此时可以删除掉之前安装的版本：

```
sudo rm -rf /usr/bin/protoc
```

继续 make plugin

继续执行 proto-gen-doc 的make，这次就可以成功了：

```
$ make
g++ -m64 -Wl,-O1 -o protoc-gen-doc mustache.o main.o qrc_protoc-gen-doc.o -lprotoc -pthread -L/usr/local/lib -lprotobuf -lQt5Core -lpthread
```

完成之后，在当前目录下就会得到生成的 protoc-gen-doc 文件。

这个文件不需要安装，直接复制到 /usr/local/bin/protoc-gen-doc 就好了，另外顺便清理一下之前安装在 /usr/bin/protoc-gen-doc 的版本：

```
sudo cp protoc-gen-doc /usr/local/bin/  
sudo rm /usr/bin/protoc-gen-doc
```

使用时发现，之前放置在 `/usr/include/` 下的include文件现在又找不到了，需要同样移动到 `/usr/local/include/` 下：

```
$ protoc --doc_out=html,index.html:../../target/contract-doc  
userService.proto  
dolphin/dolphinDescriptor.proto: File not found.  
userService.proto: Import "dolphin/dolphinDescriptor.proto" was  
not found or had errors.
```

移过去就好了

```
sudo mv /usr/include/dolphin/ /usr/local/include/
```

在CentOS6上安装

在centos 6 上安装时，遇到更多问题，记录下来备用。

1. /usr/local/bin 值得PATH路径问题

上面的方法build并安装 `protobuf-cpp` 之后，`protoc`被成功安装在

`/usr/local/bin/protoc`，直接执行 `protoc` 也可以找到，但是用jenkins就是报错找不到 `protoc`。

只好在jenkins的shell脚本里面加入一句：

```
export PATH=/usr/local/bin:$PATH
```

随后`protoc`执行时报错说找不到 `libprotoc.so.10`，但是这个文件是有build到

`/usr/local/lib` 的，只要再加一句，将 `/usr/local/lib` 加入到 `LD_LIBRARY_PATH`：

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib
```

然后继续报错，这回是 `GLIBC` 版本太久：

```
protoc-gen-doc: /lib64/libc.so.6: version `GLIBC_2.14' not found
(required by protoc-gen-doc)
protoc-gen-doc: /usr/lib64/libstdc++.so.6: version `GLIBCXX_3.4.
14' not found (required by protoc-gen-doc)
--doc_out: protoc-gen-doc: Plugin failed with status code 1.
```

安装 GLIBC 2.14 版本(参考

http://blog.csdn.net/dodo_check/article/details/9341145) :

```
wget http://mirror.bjtu.edu.cn/gnu/libc/glibc-2.14.tar.xz
xz -d glibc-2.14.tar.xz
tar xvf glibc-2.14.tar
cd glibc-2.14
mkdir build
cd build
./configure
make
make install
```

不幸报错：

```
/root/glibc-2.14/build/elf/ldconfig: Can't open configuration fi
le /usr/local/lib/glibc/etc/ld.so.conf: No such file or directory
make[1]: Leaving directory `/root/glibc-2.14'
```

使用模板定制输出

模板使用方式

protoc-gen-doc 插件支持模板，可以通过使用不同的模板来定制输出的内容和格式，命令如下：

```
protoc --doc_out=/usr/local/include/dolphin/api.mustache,index.html:../../../../target/contract-doc userService.proto
```

只是简单的将原来 `--doc_out=html,*` 中的 `html` 修改为具体的模板路径。

定制模板

默认使用的模板在 `protoc-gen-doc` 源代码根目录下的 `templates` 文件夹中：

```
$ ls
docbook.mustache  html.mustache  markdown.mustache  scalar_value_types.json
```

定制时，将 `html.mustache` 文件复制出来再修改就是，例如：

```
sudo cp html.mustache /usr/local/include/dolphin/api.mustache
```

HTML模板结构

TBD

代理

考虑一下如果有需要做gRPC的代理，该如何进行。

nghttpx

<https://nghttp2.org/documentation/nghttpx.1.html>

用于 HTTP/2, HTTP/1 和 SPDY 的反向代理。

结论：看介绍是支持了，有待进一步确认。

GCLB

GCLB 指 Google Global Load Balancer, 地址

<https://cloud.google.com/compute/docs/load-balancing/http/>

这里有一个详细的讨论 [Google Global Load Balancer support for gRPC?](#)

结论：目前 GCLB 只能支持到 Layer-3 的负载均衡，Layer-7 还在开发中，预计是 2016年底或者2017年才能发布。

grpc-proxy

gRPC proxy is a Go reverse proxy that allows for rich routing of gRPC calls with minimum overhead.

<https://github.com/mwitkow/grpc-proxy>

结论：目前还是 alpha 状态

超时

在 gRPC 中没有找到传统的超时设置，只看到在 `stub` 上有 `deadline` 的设置。但是这个设置是设置整个 `stub` 的 `deadline`，而不是单个请求。

后来通过一个 [deadline 的 issue](#) 了解到，其实可以这样来实现针对每次 RPC 请求的超时设置：

```
for (int i=0; i<100; i++) {  
    blockingStub.withDeadlineAfter(3, TimeUnit.SECONDS).doSomething();  
}
```

这里的 `.withDeadlineAfter()` 会在原有的 `stub` 基础上新建一个 `stub`，然后如果我们为每次 RPC 请求都单独创建一个有设置 `deadline` 的 `stub`，就可以实现所谓单个 RPC 请求的 `timeout` 设置。

但是代价感觉有点高，每次RPC都是一个新的 `stub`，虽然底层肯定是共享信息。

TBD：实际跑个测试试试性能，如果没有明显下降，就可以考虑采用这种方法。

Tags