**Android Perfetto trace学习笔记**

目录

[一． 简介 2](#_Toc17621)

[二． trace参数说明 7](#_Toc18399)

[三． Perfetto tracing会话的生命周期 8](#_Toc6963)

[四． 资源下载 9](#_Toc6797)

[五． 搭建Android测试环境 9](#_Toc17522)

[六． 测试指令 10](#_Toc21272)

[七． Trace配置 11](#_Toc13717)

[八． 搭建开发环境 12](#_Toc2181)

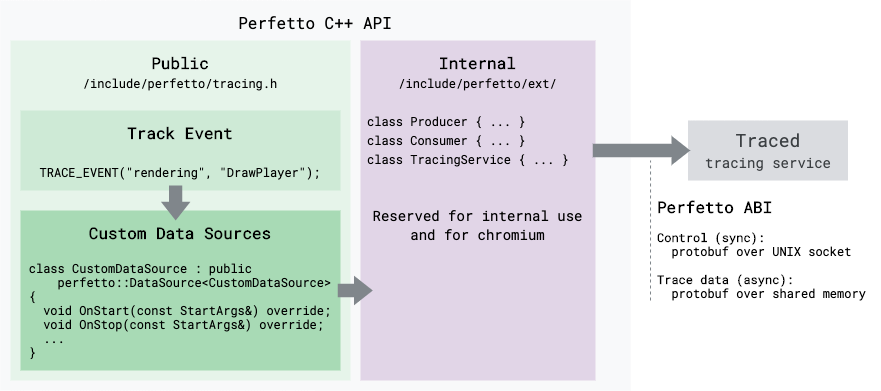
[九． 方案设计 12](#_Toc28528)

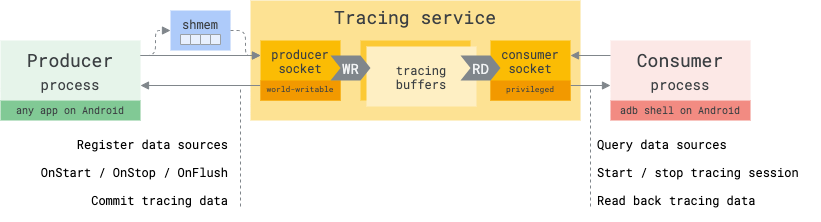
## 简介

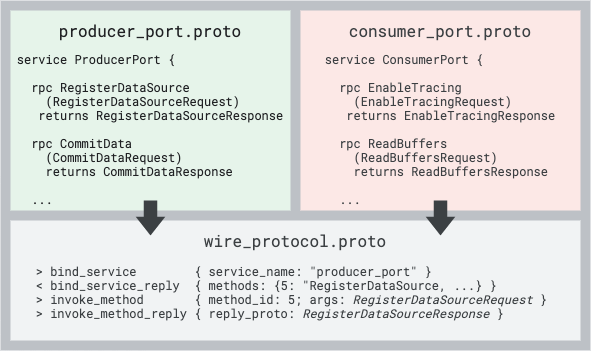
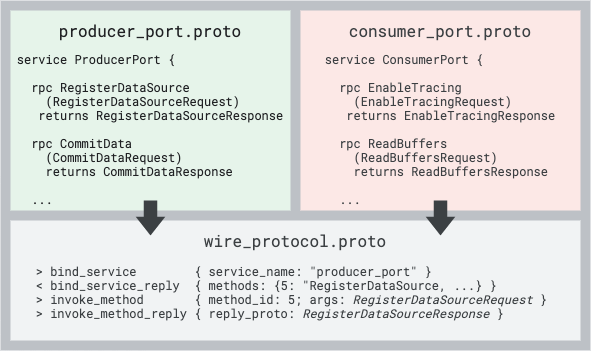
### Android 框架

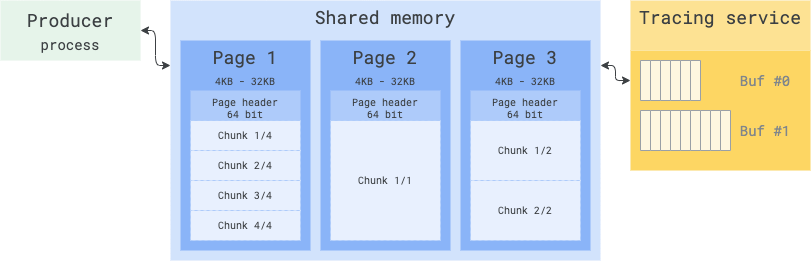


### Perfetto框架









### Profiler使用功能介绍

https://blog.csdn.net/niubitianping/article/details/72617864

### Perfetto

Perfetto 是 Android 10 中引入的全新平台级跟踪工具。这是适用于 Android、Linux 和 Chrome 的更加通用和复杂的开源跟踪项目。与 Systrace 不同，它提供数据源超集，可让您以 protobuf 编码的二进制流形式记录任意长度的跟踪记录。您可以在 [Perfetto 界面](https://links.jianshu.com/go?to=https://ui.perfetto.dev/%23!/" \t "C:\\Users\\liangweih\\AppData\\Local\\Temp\\_blank)中打开这些跟踪记录。

### Perfetto和Systrace的关系

在 Perfetto 界面中打开 Perfetto 文件和 Systrace 文件。在 Perfetto 界面中使用旧版 Systrace 查看器打开 Systrace 文件（使用 Open with legacy UI 链接）。

使用 traceconv 工具将 Perfetto 跟踪记录转换为旧版 Systrace 文本格式。

Systrace 和 Perfetto 不会收集有关应用进程中代码执行情况的详细信息。如需详细了解您的应用正在执行哪些方法及其占用了多少 CPU 资源，请使用CPU Profiler。

### Perfetto收集性能数据来源有哪些

使用 ftrace 收集内核信息

使用 atrace 收集服务和应用中的用户空间注释

使用 heapprofd 收集服务和应用的本地内存使用情况信息

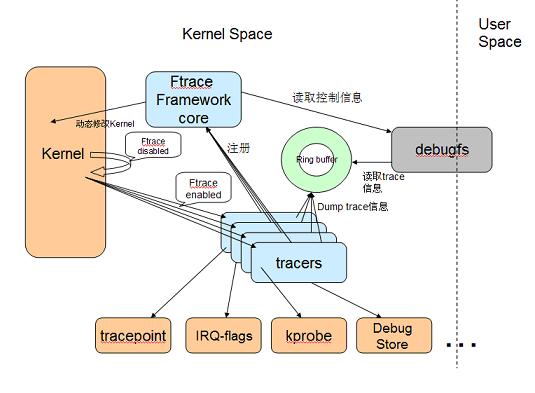
### systrace

systrace 是分析 Android 设备性能的主要工具。不过，它实际上是其他工具的封装容器：它是 atrace 的主机端封装容器，是用于控制用户空间跟踪和设置 ftrace 的设备端可执行文件，也是 Linux 内核中的主要跟踪机制。trace的核心思想是跟踪记录；

### ftrace

ftrace即linux kernel中实现的一套trace机制，可以记录各个function、event等处理的时长和调用顺序；

基于ftrace，Android拓展为systrace功能记录系统主要事件的处理path（alios 也有类似的atrace）；



Ftrace有两大组成部分，一个是framework，另外就是一系列tracer。每个tracer完成不同的功能，他们统一由framework管理。Ftrace的trace信息保存在ring buffer中，由于framework负责管理。Framework利用debugfs系统在/debugfs下建立tracing目录，并提供了一系列的控制文件。

### syslog

syslog是Linux系统默认的日志守护进程。默认的主配置文件和辅助配置文件分别是/etc/syslog.conf和/etc/sysconfig/syslog文件。通常，syslog 接受来自系统的各种功能的信息，每个信息都包括重要级。/etc/syslog.conf 文件通知 syslogd 如何根据设备和信息重要级别来报告信息。

VMA（虚拟内存区域）

dumpsys meminfo：它概述了一个进程正在使用多少种不同类型的内存。

当Android设备的内存不足时，名为的守护进程lmkd将开始终止进程​​以释放内存。设备的策略各不相同，但是一般而言，进程将按照oom\_score\_adj 得分降序被杀死（即后台应用程序和进程优先，前台进程最后）。

### Protocol Buffer

Google Protocol Buffer(简称Protobuf)是Google公司内部的混合语言数据标准，是一种轻便高效的结构化数据存储格式，可以用于结构化数据串行化，或者说序列化。可用于通信协议、数据存储等领域的语言无关（支持Java、C++、Python…）、平台无关、拓展性好的序列化结构数据格式。

Protocol Buffer 和 XML、JSON一样都是结构数据序列化的工具，但它们的数据格式有比较大的区别：

首先，Protocol Buffer 序列化之后得到的数据不是可读的字符串，而是二进制流

其次，XML 和 JSON 格式的数据信息都包含在了序列化之后的数据中，不需要任何其它信息就能还原序列化之后的数据；但使用 Protocol Buffer 需要事先定义数据的格式(.proto 协议文件)，还原一个序列化之后的数据需要使用到这个定义好的数据格式

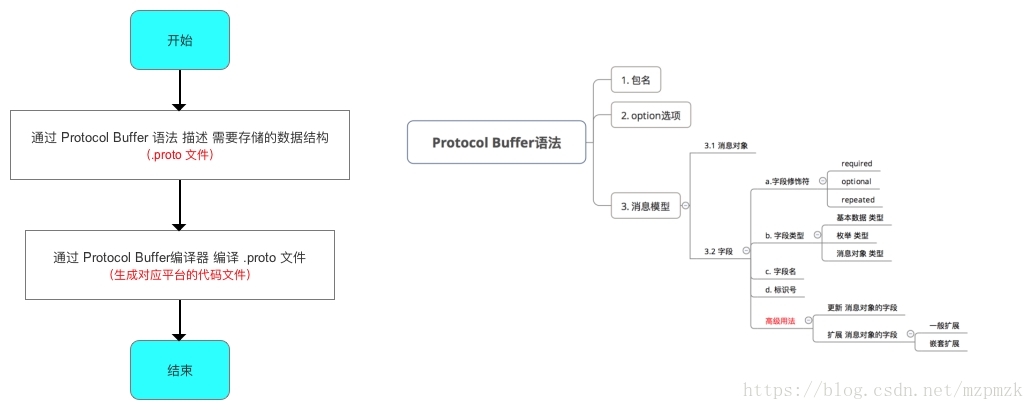
最后，在传输数据量较大的需求场景下，Protocol Buffer 比 XML、JSON 更小（3到10倍）、更快（20到100倍）、使用 & 维护更简单；而且 Protocol Buffer 可以跨平台、跨语音使用

Protocol Buffer的作用：

通过将结构化的数据（拥有多种属性）进行序列化，从而实现（内存与硬盘之间）数据存储和交换的功能

序列化： 按照 .proto 协议文件将数据结构或对象转换成二进制流的过程

反序列化：将在序列化过程中所生成的二进制流转换成数据结构或对象的过程



### .proto文件语法详解

本指南描述了怎样使用protocolbuffer语言来构造你的protocol buffer数据，包括.proto文件语法以及怎样生成.proto文件的数据访问类。

定义一个消息类型

先来看一个非常简单的例子。假设你想定义一个“搜索请求”的消息格式，每一个请求含有一个查询字符串、你感兴趣的查询结果所在的页数，以及每一页多少条查询结果。可以采用如下的方式来定义消息类型的.proto文件了：

message SearchRequest {

required string query = 1;

optional int32 page\_number = 2;

optional int32 result\_per\_page = 3;

}

SearchRequest消息格式有3个字段，在消息中承载的数据分别对应于每一个字段。其中每个字段都有一个名字和一种类型。

Ø 指定字段类型

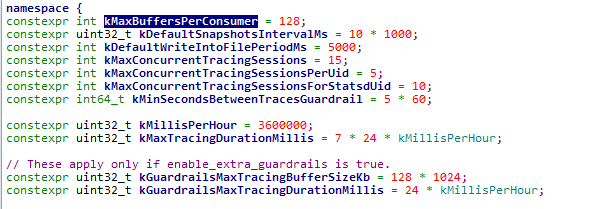
在上面的例子中，所有字段都是标量类型：两个整型（page\_number和result\_per\_page），一个string类型（query）。当然，你也可以为字段指定其他的合成类型，包括枚举（enumerations）或其他消息类型。

Ø 分配标识号

正如上述文件格式，在消息定义中，每个字段都有唯一的一个标识符。这些标识符是用来在消息的二进制格式中识别各个字段的，一旦开始使用就不能够再改 变。注：[1,15]之内的标识号在编码的时候会占用一个字节。[16,2047]之内的标识号则占用2个字节。所以应该为那些频繁出现的消息元素保留 [1,15]之内的标识号。切记：要为将来有可能添加的、频繁出现的标识号预留一些标识号。

最小的标识号可以从1开始，最大到229 - 1, or 536,870,911。不可以使用其中的[19000－19999]的标识号， Protobuf协议实现中对这些进行了预留。如果非要在.proto文件中使用这些预留标识号，编译时就会报警。

## trace参数说明



namespace {

constexpr int kMaxBuffersPerConsumer = 128;

constexpr uint32\_t kDefaultSnapshotsIntervalMs = 10 \* 1000;

constexpr int kDefaultWriteIntoFilePeriodMs = 5000;

constexpr int kMaxConcurrentTracingSessions = 15;

constexpr int kMaxConcurrentTracingSessionsPerUid = 5;

constexpr int kMaxConcurrentTracingSessionsForStatsdUid = 10;

constexpr int64\_t kMinSecondsBetweenTracesGuardrail = 5 \* 60;

constexpr uint32\_t kMillisPerHour = 3600000;

constexpr uint32\_t kMaxTracingDurationMillis = 7 \* 24 \* kMillisPerHour;

// These apply only if enable\_extra\_guardrails is true.

constexpr uint32\_t kGuardrailsMaxTracingBufferSizeKb = 128 \* 1024;

constexpr uint32\_t kGuardrailsMaxTracingDurationMillis = 24 \* kMillisPerHour;

Trace保存路径：

/data/misc/perfetto-traces/

## Perfetto tracing会话的生命周期

1. 一个或多个生产者连接到并设置它们的IPC
2. 每个生产者通过。发布一个或多个数据源[' RegisterDataSource '](/protos/perfetto/ipc/producer\_port.proto#34) ipc。在此之前，制作人不会再发生任何事情。没有tracing的违约。
3. 使用者连接到tracing服务并设置IPC通道
4. 使用者启动一个tracing会话，通过[' EnableTracing '](/protos/perfetto/ipc/consumer\_port.proto#65) ipc向服务发送一个[trace config](/docs/concepts/config.md)。
5. 服务将创建与配置中指定的相同数量的新trace缓冲区。
6. 服务遍历 [`data\_sources`]（/protos / perfetto / config / trace\_config.proto＃50）部分tracing配置：对于每个条目，如果在生产者（根据第2步中公布的内容）：
7. 服务发送一个[`SetupTracing`]（/ protos / perfetto / ipc / producer\_port.proto＃112）IPC消息，将共享内存缓冲区传递给生产者（每个仅一次）生产者）。
8. 服务发送一个[`StartDataSource`]（/ protos / perfetto / ipc / producer\_port.proto＃105）IPC消息对于每个生产者，对于在trace配置中配置的每个数据源，以及存在于生产者中（如果有）。
9. 生产者按照的指示创建一个或多个数据源实例。上一步。
10. 每个数据源实例创建一个或多个[`TraceWriter`]（/ include / perfetto / ext / tracing / core / trace\_writer.h）（通常每个线程一个）。
11. 每个`TraceWriter`都会写入一个或多个[`TracePacket`]（/ protos / perfetto / trace / trace\_packet.proto）。
12. 在这样做的时候，每个`TraceWriter`都拥有共享内存缓冲区的所有权。使用[`SharedMemoryArbiter`]（/ include / perfetto / ext / tracing / core / shared\_memory\_arbiter.h）。
13. 在编写`TracePacket`时，`TraceWriter`不可避免地会越过块边界（通常为4KB，但可以配置为更小）。
14. 发生这种情况时，`TraceWriter`将释放当前块并通过`SharedMemoryArbiter`获取一个新的块。
15. `SharedMemoryArbiter`将带外发送一个[`CommitDataRequest`]（/ protos / perfetto / ipc / producer\_port.proto＃41）到服务，要求它将共享内存缓冲区的某些块移入最终的trace缓冲区。
16. 如果一个或多个长的“ TracePacket”被分割成几个块，则为这些块中的一些可能从共享内存中消失了缓冲区并提交到最终trace缓冲区中（步骤16）。在这种情况下，“ SharedMemoryArbiter”将发送另一个“ CommitDataRequest” IPC消息请求将块数据的带外修补到最终trace中缓冲。
17. 服务将检查元组是否标识了给定的块{{ProducerID（不可伪造），WriterID，ChunkID}）仍存在于trace缓冲区，如果是，将继续对其进行修补（％检查）。
18. 消费者发送一个[`FlushRequest`]（/ protos / perfetto / ipc / consumer\_port.proto＃52）到服务，要求它在trace缓冲区中提交所有正在运行的数据。
19. 该服务依次发出一个向所有人的[`Flush`]（/ protos / perfetto / ipc / producer\_port.proto＃132）请求trace会话中涉及的生产者。
20. 生产者将全部刷新他们的TraceWriter并回复服务刷新请求。
21. 一旦服务收到来自所有人的所有刷新请求的ACK生产者（或[刷新超时]（/ protos / perfetto / ipc / consumer\_port.proto＃117）已过期）它回复消费者的“ FlushRequest”
22. 消费者可以选择发送[`DisableTracing`]（/ protos / perfetto / ipc / consumer\_port.proto＃38）IPC请求停止tracing并冻结trace缓冲区的内容。
23. 该服务将依次广播[`StopDataSource`]（/ protos / perfetto / ipc / producer\_port.proto＃110）请求每个生产者中的每个数据源。
24. 此时，消费者发出一个[`ReadBuffers`]（/protos / perfetto / ipc / consumer\_port.proto＃41）IPC请求（除非它以前确实在以下情况下将文件描述符传递给服务：通过trace配置的“ write\_into\_file”字段启用跟踪）。
25. 服务读取trace缓冲区并流所有“ TracePacket（s）”回到消费者。
26. 如果存储在trace缓冲区中的trace数据包不完整（例如，片段缺少）或标记为待处理的带外修补程序（给定的编写器）序列被中断，不再读取该序列的数据包。其他“ TraceWriter”序列的其他数据包将不受影响。
27. 消费者发送一个“ FreeBuffers”（或只是断开连接）。
28. 服务拆除了会话的所有trace缓冲区。

## 资源下载

### Android源码下载地址

<https://android.googlesource.com/platform/external/perfetto/+/f37338cdaa2126eef2a11e1a6f0cb075888c20e6>

Linux下载步骤方法：

<https://source.android.google.cn/source/downloading>

### Android源码搜索地址

<https://cs.android.com/android/platform/superproject/+/master:packages/modules/adb/>

## 搭建Android测试环境

### 安装夜神Android模拟器。

### 安装adb

1. 安装adb工具

<https://adbshell.com/downloads>

1. 下载第一个：  [ADB Kits (1364 KB)](https://adbshell.com/upload/adb.zip)(adb.exe AdbWinApi.dll AdbWinUsbApi.dll)
2. 解压，解压路径不能有中文。
3. 在环境变量中添加解压文件路径。

### 用adb连接安卓模拟器

国产出了很多速度比较快的安卓模拟器，不过每次ide重启后需要用adb命令重连模拟器

adb connect 127.0.0.1

这句命令默认会连5555端口，谷歌官方模拟器就是用这个端口，但是这些国产模拟器用的端口却不一样

windows下的

夜神模拟器 adb connect 127.0.0.1:62001

逍遥模拟器 adb connect 127.0.0.1:21503

## 测试指令

### 查看是否有Android设备

adb devices

### 打开perfetto的设置界面

adb shell am start com.android.traceur/com.android.traceur.MainActivity

方法2：打开Settings->System->Developer options->System Tracing

界面介绍各个选项的作用：

Record trace开启和关闭trace功能

Trace debuggable applications开启和关闭支持debuggable的应用的自定义trace

Categories选择Trace Categories

Restore default categories回复默认的Trace Categories

Per-CPU buffer size每一个CPU的buffer大小，一般情况下不用改大 ，如果发现Trace有丢失或者在userdebug手机上抓取trace，最好增大一下，避免trace的丢失。

Clear saved traces清除手机里已经存储的trace文件

Long traces一般我们抓的trace，都只会记录开启Record trace之后30s的trace。对于需要长时间跟踪的，可以开启这个选项并且配置下方的Maximum long trace size和Maximum long trace duration

### 点击Record trace按钮

### 复现性能有问题的场景

### 下拉状态栏，点击关闭trace，等待trace文件保存

### 导出trace文件

adb pull data/local/traces E:\TestData

### 打开分析网站并打开trace文件进行分析

https://ui.perfetto.dev/#!/viewer  查看perfetto数据的网站

## Trace配置

使用传统的 ftrace 需要如下几个步骤：

* 选择一种 tracer
* 使能 ftrace
* 执行需要 trace 的应用程序，比如需要跟踪 ls，就执行 ls
* 关闭 ftrace
* 查看 trace 文件

ftrace 支持两大类 tracer：传统 tracer 和 Non-Tracer Tracer

ftrace 的输出信息主要保存在 3 个文件中。

* Trace，该文件保存 ftrace 的输出信息，其内容可以直接阅读。
* latency\_trace，保存与 trace 相同的信息，不过组织方式略有不同。主要为了用户能方便地分析系统中有关延迟的信息。
* trace\_pipe 是一个管道文件，主要为了方便应用程序读取 trace 内容。算是扩展接口吧。

从 2.6.30 开始，ftrace 还支持几种 Non-tracer tracer，所谓 Non-tracer tracer 主要包括以下几种：

* Max Stack Tracer
* Profiling (branches / unlikely / likely / Functions)
* Event tracing

A simple trace config looks like this:

```protobuf

duration\_ms: 10000

buffers {

size\_kb: 65536

fill\_policy: RING\_BUFFER

}

data\_sources {

config {

name: "linux.ftrace"

target\_buffer: 0

ftrace\_config {

ftrace\_events: "sched\_switch"

ftrace\_events: "sched\_wakeup"

}

}

}

````

## 搭建开发环境

安装Android studio环境，包括Java、JDK

## 方案设计