# 背景

互联网上的机器大都通过TCP/IP协议相互访问，但TCP/IP只是往远端发送了一段二进制数据，为了建立服务还有很多问题需要抽象：

数据以什么格式传输？不同机器间，网络间可能是不同的字节序，直接传输内存数据显然是不合适的；随着业务变化，数据字段往往要增加或删减，怎么兼容前后不同版本的格式？

一个TCP连接可以被多个请求复用以减少开销么？多个请求可以同时发往一个TCP连接么?

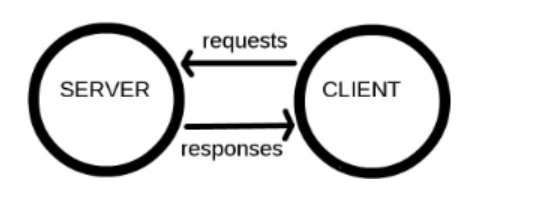
如何管理和访问很多机器？

连接断开时应该干什么？

万一server不发送回复怎么办？

…

RPC可以解决这些问题，它把网络交互类比为“client访问server上的函数”：client向server发送request后开始等待，直到server收到、处理、回复client后，client又再度恢复并根据response做出反应。



我们来看看上面的一些问题是如何解决的：

数据需要序列化，protobuf在这方面做的不错。用户填写protobuf::Message类型的request，RPC结束后，从同为protobuf::Message类型的response中取出结果。protobuf有较好的前后兼容性，方便业务调整字段。http广泛使用json作为序列化方法。

用户无需关心连接如何建立，但可以选择不同的连接方式：短连接，连接池，单连接。

大量机器一般通过命名服务被发现，可基于DNS, ZooKeeper, etcd等实现。在百度内，我们使用BNS (Baidu Naming Service)。brpc也提供“list://“和"file://”。用户可以指定负载均衡算法，让RPC每次选出一台机器发送请求，包括: round-robin, randomized, consistent-hashing(murmurhash3 or md5)和 locality-aware.

连接断开时可以重试。

如果server没有在给定时间内回复，client会返回超时错误。

**RPC的应用场景：**

几乎所有的网络交互。

RPC不是万能的抽象，否则我们也不需要TCP/IP这一层了。但是在我们绝大部分的网络交互中，RPC既能解决问题，又能隔离更底层的网络问题。

对于RPC常见的质疑有：

我的数据非常大，用protobuf序列化太慢了。首先这可能是个伪命题，你得用profiler证明慢了才是真的慢，其次很多协议支持携带二进制数据以绕过序列化。

我传输的是流数据，RPC表达不了。事实上brpc中很多协议支持传递流式数据，包括http中的ProgressiveReader, h2的streams, streaming rpc, 和专门的流式协议RTMP。

我的场景不需要回复。简单推理可知，你的场景中请求可丢可不丢，可处理也可不处理，因为client总是无法感知，你真的确认这是OK的？即使场景真的不需要，我们仍然建议用最小的结构体回复，因为这不大会是瓶颈，并且追查复杂bug时可能是很有价值的线索。

# 概述

参考：<https://blog.csdn.net/wxj1992/category_11267957.html>

brpc是用c++语言编写的工业级RPC框架，常用于搜索、存储、机器学习、广告、推荐等高性能系统。

## 特点

### 优点

1、更友好的接口

只有三个(主要的)用户类: Server, Channel, Controller, 分别对应server端，client端，参数集合. 你不必推敲诸如"如何初始化XXXManager”, “如何组合各种组件”, “XXXController的XXXContext间的关系是什么”。要做的很简单:

建服务? 包含brpc/server.h并参考注释或示例.

访问服务? 包含brpc/channel.h并参考注释或示例.

调整参数? 看看brpc/controller.h. 注意这个类是Server和Channel共用的，分成了三段，分别标记为Client-side, Server-side和Both-side methods。

我们尝试让事情变得更加简单，以命名服务为例，在其他RPC实现中，你也许需要复制一长段晦涩的代码才可使用，而在brpc中访问BNS可以这么写"bns://node-name"，DNS是"http://domain-name"，本地文件列表是"file:///home/work/server.list"，相信不用解释，你也能明白这些代表什么。

2、使服务更加可靠

brpc在百度内被广泛使用:

map-reduce服务和table存储

高性能计算和模型训练

各种索引和排序服务

….

它是一个经历过考验的实现。

brpc特别重视开发和维护效率, 你可以通过浏览器或curl查看server内部状态, 分析在线服务的cpu热点, 内存分配和锁竞争, 通过bvar统计各种指标并通过/vars查看。

3、更好的延时和吞吐

虽然大部分RPC实现都声称“高性能”，但数字仅仅是数字，要在广泛的场景中做到高性能仍是困难的。为了统一百度内的通信架构，brpc在性能方面比其他RPC走得更深。

对不同客户端请求的读取和解析是完全并发的，用户也不用区分”IO线程“和”处理线程"。其他实现往往会区分“IO线程”和“处理线程”，并把fd（对应一个客户端）散列到IO线程中去。当一个IO线程在读取其中的fd时，同一个线程中的fd都无法得到处理。当一些解析变慢时，比如特别大的protobuf message，同一个IO线程中的其他fd都遭殃了。虽然不同IO线程间的fd是并发的，但你不太可能开太多IO线程，因为这类线程的事情很少，大部分时候都是闲着的。如果有10个IO线程，一个fd能影响到的”其他fd“仍有相当大的比例（10个即10%，而工业级在线检索要求99.99%以上的可用性）。这个问题在fd没有均匀地分布在IO线程中，或在多租户(multi-tenancy)环境中会更加恶化。在brpc中，对不同fd的读取是完全并发的，对同一个fd中不同消息的解析也是并发的。解析一个特别大的protobuf message不会影响同一个客户端的其他消息，更不用提其他客户端的消息了。更多细节看这里。

对同一fd和不同fd的写出是高度并发的。当多个线程都要对一个fd写出时（常见于单连接），第一个线程会直接在原线程写出，其他线程会以wait-free的方式托付自己的写请求，多个线程在高度竞争下仍可以在1秒内对同一个fd写入500万个16字节的消息。更多细节看这里。

尽量少的锁。高QPS服务可以充分利用一台机器的CPU。比如为处理请求创建bthread, 设置超时, 根据回复找到RPC上下文, 记录性能计数器都是高度并发的。即使服务的QPS超过50万，用户也很少在contention profiler)中看到框架造成的锁竞争。

服务器线程数自动调节。传统的服务器需要根据下游延时的调整自身的线程数，否则吞吐可能会受影响。在brpc中，每个请求均运行在新建立的bthread中，请求结束后线程就结束了，所以天然会根据负载自动调节线程数。

### 缺点

## 应用场景

你可以使用它：

1、搭建能在一个端口支持多协议的服务, 或访问各种服务

restful http/https, h2/gRPC。使用brpc的http实现比libcurl方便多了。从其他语言通过HTTP/h2+json访问基于protobuf的协议.

redis和memcached, 线程安全，比官方client更方便。

rtmp/flv/hls, 可用于搭建流媒体服务.

hadoop\_rpc(可能开源)

支持rdma(即将开源)

支持thrift , 线程安全，比官方client更方便

各种百度内使用的协议: baidu\_std, streaming\_rpc, hulu\_pbrpc, sofa\_pbrpc, nova\_pbrpc, public\_pbrpc, ubrpc和使用nshead的各种协议.

基于工业级的RAFT算法实现搭建高可用分布式系统，已在braft开源。

2、Server能同步或异步处理请求。

3、Client支持同步、异步、半同步，或使用组合channels简化复杂的分库或并发访问。

4、通过http界面调试服务, 使用cpu, heap, contention profilers.

获得更好的延时和吞吐.

5、把你组织中使用的协议快速地加入brpc，或定制各类组件, 包括命名服务 (dns, zk, etcd), 负载均衡 (rr, random, consistent hashing)

# 安装配置

# 原理

## bvar

## bthread

## M:N线程模型

## Client端执行流程

## Server端执行流程

## 并发读写TCP连接上的数据

## 内存管理

## 实时监控

## 基础库

# 应用

## Spider支持bthread

### 概述

spider是基于percona的，原生的服务模型是前端一个连接对应spider一个线程。

当并发较大时，线程的切换开销代价不容忽视。对此，可以引入更轻量级的bthread。

### 总体设计

主要有三方面的改动：

1. 和前端client的交互修改
2. sql层锁相关的修改
3. innodb层锁相关的修改

#### 和前端client的交互

1、增加bthread的启动模式，一个connection对应一个bthread

新增一种工作模式SCHEDULER\_THREAD\_CO：

enum scheduler\_types {

#ifdef WITH\_SPIDER\_STORAGE\_ENGINE

SCHEDULER\_THREAD\_CO = 0,

SCHEDULER\_ONE\_THREAD\_PER\_CONNECTION,

#elif

SCHEDULER\_ONE\_THREAD\_PER\_CONNECTION =0,

#endif

SCHEDULER\_NO\_THREADS,

SCHEDULER\_THREAD\_POOL,

SCHEDULER\_TYPES\_COUNT

};

对应的处理逻辑是：

Connection\_handler\_manager::init()

|-->Per\_bthread\_connection\_handler()

|--|-->Per\_bthread\_connection\_handler::add\_connection

|--|--|-->bthread\_handle\_connection

关键代码：

Per\_bthread\_connection\_handler::add\_connection

|-->启动bthread，工作函数是bthread\_handle\_connection

|--|-->do\_command(thd) 完成sql请求

2、和mysql client的socket有网络交互的时候要使用非阻塞io，bthread能主动切走并在对应的fd上有event的时候自动切回来

核心思想是利用bthread\_fd\_timedwait

关键代码：

int vio\_io\_wait(Vio \*vio, enum enum\_vio\_io\_event event, int timeout) {

if (timeout == 0) {

return vio\_io\_wait\_old(vio, event, timeout);

}

return vio\_io\_wait\_bthread(vio, event, timeout);

}

#### sql层锁相关修改

1、将mysql mutex，condition variable相关的宏定义用bthread mutex，condition variable替换

使用CO\_MUTEX条件编译，将原来pthread相关函数换成bthread

关键代码示例：

/////////////////////////////////////////////

#ifdef \_WIN32

typedef CRITICAL\_SECTION native\_mutex\_t;

typedef int native\_mutexattr\_t;

#elif defined(CO\_MUTEX)

typedef bthread\_mutex\_t native\_mutex\_t;

typedef bthread\_mutexattr\_t native\_mutexattr\_t;

#else

typedef pthread\_mutex\_t native\_mutex\_t;

typedef pthread\_mutexattr\_t native\_mutexattr\_t;

#endif

​

/////////////////////////////////////////////

#elif defined(CO\_MUTEX)

return bthread\_mutex\_init\lock\unlock\try\_lock\destroy(mutex);

#else

return pthread\_mutex\_init\lock\unlock\try\_lock\destroy(mutex);

#endif

/////////////////////////////////////////////

#ifdef \_WIN32

typedef CONDITION\_VARIABLE native\_cond\_t;

#elif defined(CO\_MUTEX)

typedef bthread\_cond\_t native\_cond\_t;

#else

typedef pthread\_cond\_t native\_cond\_t;

#endif

/////////////////////////////////////////////

​

#### ​innodb锁相关的修改

1. mutex/condition相关操作换成bthread的函数

举例：

#ifdef \_WIN32

DeleteCriticalSection(mutex);

return 0;

#elif defined(CO\_MUTEX)

return bthread\_mutex\_destroy(mutex);

#else

return pthread\_mutex\_destroy(mutex);

#endif

1. 类似pthread-id，给出bthread-id，否则debug模式下，很多断言会失败

// bthread or pthread self

static bthread\_t tdsql\_thread\_self() {

bthread\_t bid = bthread\_self();

if (bid != BTHD\_INVALID\_BTHREAD) {

return bid;

}

return pthread\_self();

}

3、thread local的实现方式

使用线程时，可以用关键字thread local指定变量是线程局部变量。切换到bthread后，继续使用这种方式会出现无法预料的情况，甚至是crash；

需要改为bthread\_getspecific + bthread\_setspecific的方式，去控制bthread local变量。

举例：

1）增加bthread线程变量定义

#ifndef CO\_MUTEX

static thread\_local const trx\_t \*trx\_first\_latched\_trx = nullptr;

static thread\_local int32\_t trx\_latched\_count = 0;

static thread\_local bool trx\_allowed\_two\_latches = false;

#endif

#if 0

static bthread\_key\_t trx\_first\_latched\_trx\_key;

static bthread\_key\_t trx\_latched\_count\_key;

static bthread\_key\_t trx\_allowed\_two\_latches\_key;

2）修改thread local变量使用方式

void trx\_before\_mutex\_enter(const trx\_t \*trx, bool first\_of\_two) {

#ifdef CO\_MUTEX

trx\_t \*\*trx\_first\_latched\_trx = static\_cast<trx\_t \*\*>(bthread\_getspecific(trx\_first\_latched\_trx\_key));

if (trx\_first\_latched\_trx == nullptr) {

trx\_first\_latched\_trx = new (trx\_t\*);

\*trx\_first\_latched\_trx = nullptr;

}

int32\_t \*trx\_latched\_count = static\_cast<int32\_t\*>(bthread\_getspecific(trx\_latched\_count\_key));

if (trx\_latched\_count == nullptr) {

trx\_latched\_count = new int32\_t;

\*trx\_latched\_count = 0;

}

bool \*trx\_allowed\_two\_latches = static\_cast<bool\*>(bthread\_getspecific(trx\_allowed\_two\_latches\_key));

if (trx\_allowed\_two\_latches == nullptr) {

trx\_allowed\_two\_latches = new bool;

\*trx\_allowed\_two\_latches = false;

}

if (0 == (\*trx\_latched\_count)++) {

ut\_a(\*trx\_first\_latched\_trx == nullptr);

\*trx\_first\_latched\_trx = const\_cast<trx\_t\*>(trx);

bthread\_setspecific(trx\_first\_latched\_trx\_key, trx\_first\_latched\_trx);

if (first\_of\_two) {

\*trx\_allowed\_two\_latches = true;

bthread\_setspecific(trx\_allowed\_two\_latches\_key, trx\_allowed\_two\_latches);

}

} else {

ut\_a(!first\_of\_two);

if (!locksys::owns\_exclusive\_global\_latch()) {

ut\_a(\*trx\_allowed\_two\_latches);

ut\_a(\*trx\_latched\_count == 2);

ut\_a((\*trx\_first\_latched\_trx)->lock.wait\_lock == nullptr);

ut\_a(\*trx\_first\_latched\_trx != trx);

/\* This is not very safe, because to read trx->lock.wait\_lock we

should already either latch trx->mutex (which we don't) or shard with

trx->lock.wait\_lock. But our claim is precisely that we have latched

this shard, and we want to check that here. \*/

ut\_a(trx->lock.wait\_lock != nullptr);

ut\_a(locksys::owns\_lock\_shard(trx->lock.wait\_lock));

}

}

bthread\_setspecific(trx\_latched\_count\_key, trx\_latched\_count);

#else

if (0 == trx\_latched\_count++) {

ut\_a(trx\_first\_latched\_trx == nullptr);

trx\_first\_latched\_trx = trx;

if (first\_of\_two) {

trx\_allowed\_two\_latches = true;

}

} else {

ut\_a(!first\_of\_two);

if (!locksys::owns\_exclusive\_global\_latch()) {

ut\_a(trx\_allowed\_two\_latches);

ut\_a(trx\_latched\_count == 2);

ut\_a(trx\_first\_latched\_trx->lock.wait\_lock == nullptr);

ut\_a(trx\_first\_latched\_trx != trx);

/\* This is not very safe, because to read trx->lock.wait\_lock we

should already either latch trx->mutex (which we don't) or shard with

trx->lock.wait\_lock. But our claim is precisely that we have latched

this shard, and we want to check that here. \*/

ut\_a(trx->lock.wait\_lock != nullptr);

ut\_a(locksys::owns\_lock\_shard(trx->lock.wait\_lock));

}

}

#endif

}

参考：<https://blog.csdn.net/u010445301/article/details/111322569>

### 遗留问题

1、存在内存泄漏问题：压力较大时，资源申请get\_resource的速度大于return\_resource的速度；

2、目前是工作线程改为了bthread，还有部分后台线程，也可以继续改造为bthread；

3、改造后性能没有达到预期，比改造前的pthread模型性能还有减弱。