# 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

## 对比

### bRPC vs TCP

bRPC与TCP在通信协议层面有着不同的设计和应用场景，各自都有其独特的优势。

bRPC的优势主要体现在以下几个方面：

易用性：bRPC提供了丰富的接口和特性，使得开发者可以更方便地进行远程过程调用（RPC）。它支持多种序列化协议，如Protocol Buffers，使得数据交换更加高效和灵活。

高性能：bRPC在内部实现上做了大量的优化，包括连接复用、请求合并、负载均衡等，从而提高了系统的吞吐量和响应速度。

可靠性：bRPC提供了多种容错机制，如超时重试、失败自动切换等，保证了系统的稳定性和可靠性。

安全性：bRPC支持TLS加密通信，可以有效保护数据的安全性和隐私性。

TCP的优势则主要体现在以下几个方面：

通用性：TCP是互联网协议族中的传输层协议，广泛应用于各种网络应用中。它提供了可靠的、面向连接的、字节流的传输服务，适用于各种数据传输场景。

灵活性：TCP协议本身并不限制上层应用的数据格式和通信方式，因此开发者可以根据自己的需求定制通信协议和数据处理逻辑。

成熟稳定：TCP协议经过多年的发展和优化，已经非常成熟和稳定。在大多数情况下，使用TCP进行通信可以获得较好的性能和可靠性。

在选择使用TCP还是bRPC时，需要根据具体的应用场景和需求来决定。

* 适合使用TCP的场景包括：

需要进行大量数据传输的场景，如文件传输、流媒体传输等。

对通信协议有自定义需求的场景，如某些特殊的应用协议或通信方式。

网络环境复杂多变，需要灵活应对的场景。

* 适合使用bRPC的场景包括：

需要进行远程过程调用的场景，如微服务架构中的服务间通信。

对性能和可靠性有较高要求的场景，如金融交易、在线游戏等。

需要保护数据安全和隐私的场景，如使用TLS加密通信。

综上所述，bRPC和TCP各有其优势和适用场景。在选择使用哪种通信方式时，需要综合考虑应用需求、网络环境、性能要求等因素。

### bthread VS pthread

bRPC中的bthread是专门为bRPC框架设计的一套轻量级线程库，它结合了协程（coroutine）和线程池的概念，以提供更高效、更灵活的并发处理能力。以下是bthread的原理和优势，以及与Linux的pthread的比较：

* bthread原理

bthread不是传统意义上的线程，而是基于用户态轻量级线程（LWT）的实现。每个bthread都映射到一个真实的操作系统线程上，但多个bthread可以共享同一个操作系统线程的执行上下文。通过协作式调度，bthread可以在用户态实现无锁切换，从而避免了内核态切换的开销。

* bthread优势

高效性：由于bthread在用户态进行切换，避免了内核态的上下文切换开销，因此切换速度更快。同时，通过合理的线程池管理，可以有效减少线程创建和销毁的开销。

轻量级：bthread相比传统线程更加轻量级，占用的资源更少。这使得可以创建更多的并发任务，提高系统的并发处理能力。

易用性：bthread的API设计简洁明了，易于理解和使用。同时，bRPC框架也提供了丰富的功能和工具，方便开发者进行高性能的RPC通信。

灵活性：bthread支持异步编程模型，可以更好地处理IO密集型任务。同时，通过协作式调度和事件驱动机制，可以实现更灵活的并发控制。

* 与Linux pthread的比较

切换开销：pthread的切换涉及内核态的上下文切换，开销相对较大。而bthread在用户态进行切换，切换速度更快。

资源占用：pthread创建和销毁线程的开销较大，每个线程都需要占用一定的系统资源。而bthread更加轻量级，可以创建更多的并发任务。

编程模型：pthread采用传统的线程编程模型，需要开发者手动管理线程的创建、销毁和同步等问题。而bthread结合了协程和线程池的概念，提供了更简洁、更高效的编程模型。

适用场景：pthread适用于需要精细控制线程行为的场景，如复杂的并发计算任务。而bthread更适用于IO密集型任务和高性能的RPC通信场景。

* bthread适用场景：

高性能的RPC通信，特别是IO密集型任务。

需要大量并发处理的场景，如微服务架构中的服务调用。

追求高效、轻量级并发控制的场景。

* pthread适用场景：

需要精细控制线程行为的场景，如复杂的并发计算任务。

需要使用系统级线程同步原语的场景。

传统的多线程编程场景，如多线程服务器等。

综上所述，bthread和pthread各有其优势和适用场景。在选择使用哪种线程库时，需要根据具体的应用需求和场景来决定。

## 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

参考：

<https://github.com/ronaldo8210/brpc_source_code_analysis>

bRPC收到请求的处理过程：

<https://blog.csdn.net/wxj1992/article/details/97687584>

高性能RPC框架BRPC核心机制分析：

<https://zhuanlan.zhihu.com/p/113427004>

bRPC的一些高并发机制：<https://blog.csdn.net/asdasdkhs/article/details/128854619>

## bvar

参考：

brpc中的bvar介绍：<https://zhuanlan.zhihu.com/p/421149320>

<https://zhuanlan.zhihu.com/p/640952736>

<https://blog.csdn.net/wxj1992/article/details/105134641>

### 背景

### 概述

### 实现

#### 基类Variable

#### Variable的主要派生类

##### bvar::Reducer

##### bvar::IntRecorder

##### bvar::Status

##### bvar::PassiveStatus

##### bvar::Window

##### bvar::PerSecond

##### bvar::LatencyRecorder

LatencyRecorder应该是bvar最常用的一个结构了，他用来存储延迟，并提供qps/latency\_avg/latency\_p99等多个统计值。

为了统计p99等信息，LatencyRecorder需要一个结构来记录数据分布，这个也是在读LatencyRecorder之前所需要了解的最后一个结构，叫做Percentile

Percentile中也包含一个combiner和一个sampler，其中Combiner就是之前看到的AgentCombiner，而Sampler也是之前看到的ReducerSampler，唯一有不同的就是Element的类型不同，这里的Element为PercentileSamples

每个PercentileSamples包含32个PercentileInterval，每个PercentileInterval包含若干个Sample，每个Sample就是用户给的Latency，Sample的数量由PercentileInterval的模版类型所指定（减少动态内存分配次数）

之所以每个PercentileSamples包含32个PercentileInterval，是因为LatencyRecorder里通过uint32\_t来记录Latency，并且会根据log(latency)的值来计算桶的位置，所以就最多需要32个桶。

在TDSQL3.0中，获取95%的耗时采用bvar实现：

bvar::LatencyRecorder \*QueryLatencyInstance() {

static bvar::LatencyRecorder \*query\_latency =

new bvar::LatencyRecorder("query\_latency");

return query\_latency;

}

double avg\_cost\_calculate() {

return QueryLatencyInstance()->latency();

}

double sb\_percentile\_calculate(double percent) {

return QueryLatencyInstance()->latency\_percentile(percent/100);

}

static int show\_time\_cost\_95(THD \*, SHOW\_VAR \*var, char \*buff) {

var->type = SHOW\_LONGLONG;

var->value = buff;

long long \*value = reinterpret\_cast<long long \*>(buff);

\*value = (long long)sb\_percentile\_calculate(95);

return 0;

}

static int show\_time\_cost\_99(THD \*, SHOW\_VAR \*var, char \*buff) {

var->type = SHOW\_LONGLONG;

var->value = buff;

long long \*value = reinterpret\_cast<long long \*>(buff);

\*value = (long long)sb\_percentile\_calculate(99);

return 0;

}

static int show\_time\_cost\_avg(THD \*, SHOW\_VAR \*var, char \*buff) {

var->type = SHOW\_LONGLONG;

var->value = buff;

long long \*value = reinterpret\_cast<long long \*>(buff);

\*value = (long long)avg\_cost\_calculate();

return 0;

}

### bvar的采集

## bthread

### 背景

### 概述

### 原理

### thread-local

### 使用

## M:N线程模型

线程模型：<https://github.com/apache/brpc/blob/master/docs/cn/threading_overview.md>

### bthread基础

#### 线程执行协程与线程调用函数的不同

一个pthread系统线程执行一个函数时，需要在pthread的线程栈上为函数创建栈帧，函数的形式参数和局部变量都分配在栈帧内。函数执行完毕后，按逆序销毁函数局部变量，再销毁栈帧。假设有一个线程A开始执行下面的foo函数：

void bar(int m) {

// 执行点1

// ...

}

void foo(int a, int b) {

int c = a + b;

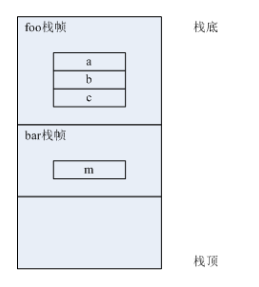
bar(c);

// 执行点2

// ...

}

执行到foo函数中的执行点1时，线程A的栈帧如下图所示：



线程A从bar()函数返回，执行到执行点2时，先销毁bar()函数的形参m，再销毁bar()的栈帧，从foo()函数返回后，先销毁局部变量c，接着销毁形参b、a，最后销毁foo()的栈帧。

像上述这种在foo()函数内调用bar()函数的过程，必须等到bar()函数return后，foo()函数才从bar()函数的返回点恢复执行。

一个协程可以看做是一个单独的任务，相应的也有一个任务处理函数。对协程来说最重要的两个操作是yield和resume（yield和resume的实现见下文描述），yield是指一个正在被pthread系统线程执行的协程被挂起，让出cpu的使用权，pthread继续去执行另一个协程的任务函数，从协程角度看是协程中止了运行，从系统线程角度看是pthread继续在运行；resume是指一个被中止的协程的任务函数重新被pthread执行，恢复执行点为上一次yield操作的返回点。

有了yield和resume这两个原语，可以实现pthread线程执行流在不同函数间的跳转，只需要将函数作为协程的任务函数即可。一个线程执行一个协程A的任务处理函数taskFunc\_A时，如果想要去执行另一个协程B的任务处理函数taskFunc\_B，不必等到taskFunc\_A执行到return语句，可以在taskFunc\_A内执行一个yield语句，然后线程执行流可以从taskFunc\_A中跳出，去执行taskFunc\_B。如果想让taskFunc\_A恢复执行，则调用一个resume语句，让taskFunc\_A从yield语句的返回点处开始继续执行，并且taskFunc\_A的执行结果不受yield的影响。

说明：从上述的描述可知，如果进行协程化编程，其实是需要在中间执行过程中考虑多个协程的竞争问题的，并且做出相应的处理。

#### 协程的原理与实现方式

协程有三个组成要素：一个任务函数，一个存储寄存器状态的结构，一个私有栈空间（通常是malloc分配的一块内存，或者static静态区的一块内存）。

协程被称作**用户级线程**就是因为协程有其**私有的栈空间**，pthread系统线程调用一个普通函数时，函数的栈帧、形参、局部变量都分配在pthread线程的栈上，而pthread执行一个协程的任务函数时，协程任务函数的栈帧、形参、局部变量都分配在协程的私有栈上。

yield和resume有多种实现方式，可以使用posix的ucontext，boost的fcontext，或者直接用汇编实现。下面用ucontext讲述下如何实现协程的yield和resume：

1、posix定义的ucontext数据结构如下：

typedef struct ucontext

{

unsigned long int uc\_flags;

// uc\_link指向的ucontext是后继协程的ucontext，当前协程的任务函数return后，会自动将后继协程

// 的ucontext缓存的寄存器值加载到cpu的寄存器中，cpu会去执行后继协程的任务函数（可能是从函数

// 入口点开始执行，也可能是从函数yield调用的返回点恢复执行）。

struct ucontext \*uc\_link;

// 协程私有栈。

stack\_t uc\_stack;

// uc\_mcontext结构用于缓存协程yield时，cpu各个寄存器的当前值。

mcontext\_t uc\_mcontext;

\_\_sigset\_t uc\_sigmask;

} ucontext\_t;

2、ucontext的api接口有如下四个：

int getcontext(ucontext\_t \*ucp)

将cpu的各个寄存器的当前值存入当前正在被cpu执行的协程A的ucontext\_t的uc\_mcontext结构中。重要的寄存器有栈底指针寄存器、栈顶指针寄存器、协程A的任务函数下一条将被执行的语句的指令指针寄存器等。

int setcontext(const ucontext\_t \*ucp)

将一个协程A的ucontext\_t的uc\_mcontext结构中缓存的各种寄存器的值加载到cpu的寄存器中，cpu可以根据栈底指针寄存器、栈顶指针寄存器定位到该协程A的私有栈空间，根据指令指针寄存器定位到协程A的任务函数的执行点（可能为函数入口点也可能为函数yield调用的返回点），从而cpu可以去执行协程A的任务函数，并将函数执行过程中产生的局部变量等分配在协程A的私有栈上。

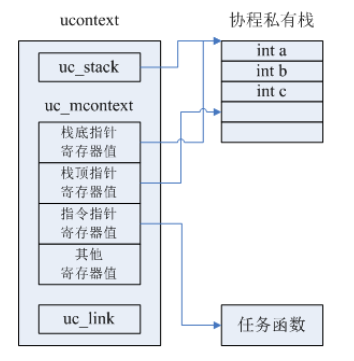
void makecontext(ucontext\_t \*ucp, void (\*func)(), int argc, ...)

指定一个协程的任务函数func以及func的argc等参数。

int swapcontext(ucontext\_t \*oucp, ucontext\_t \*ucp)

相当于getcontext(oucp) + setcontext(ucp)的原子调用，将cpu寄存器的当前值存入oucp指向的ucontext\_t的uc\_mcontext结构中，并将ucp指向的ucontext\_t的uc\_mcontext结构中缓存的寄存器值加载到cpu的寄存器上，目的是让当前协程yield，让ucp对应的协程start或resume。

用ucontext实现的一个协程的内存布局如下图所示：



#### 系统线程执行多个协程时的内存布局变化过程

下面通过一个协程示例程序，展现pthread系统线程执行多个协程时的内存变化过程：

static ucontext\_t ctx[3];

static void func\_1(void) {

int a;

// 执行点2，协程1在这里yield，pthread线程恢复执行协程2的任务函数，即令协程2 resume。

swapcontext(&ctx[1], &ctx[2]);

// 执行点4，协程1从这里resume恢复执行。

// func\_1 return后，由于ctx[1].uc\_link = &ctx[0]，将令main函数resume。

}

static void func\_2(void) {

int b;

// 协程2在这里yield，pthread线程去执行协程1的任务函数func\_1。

swapcontext(&ctx[2], &ctx[1]);

// 执行点3，协程2从这里resume恢复执行。

// func\_2 return后，由于ctx[2].uc\_link = &ctx[1]，将令协程1 resume。

}

int main(int argc, char \*\*argv) {

// 定义协程1和协程2的私有栈。

// 在这个程序中，协程1和协程2都在main函数return之前执行完成，

// 所以将协程私有栈内存区定义为main函数的局部变量是安全的。

char stack\_1[1024] = { 0 };

char stack\_2[1024] = { 0 };

// 初始化协程1的ucontext\_t结构ctx[1]。

getcontext(&ctx[1]);

// 在ctx[1]结构中指定协程1的私有栈stack\_1。

ctx[1].uc\_stack.ss\_sp = stack\_1;

ctx[1].uc\_stack.ss\_size = sizeof(stack\_1);

// ctx[0]用于存储执行main函数所在线程的cpu的各个寄存器的值，

// 下面语句的作用是，当协程1的任务函数return后，将ctx[0]中存储的各寄存器的值加载到cpu的寄存器中，

// 也就是pthread线程从main函数之前的yield调用的返回处继续执行。

ctx[1].uc\_link = &ctx[0];

// 指定协程1的任务函数为func\_1。

makecontext(&ctx[1], func1, 0);

// 初始化协程2的ucontext\_t结构ctx[2]。

getcontext(&ctx[2]);

// 在ctx[2]结构中指定协程2的私有栈stack\_2。

ctx[2].uc\_stack.ss\_sp = stack\_2;

ctx[2].uc\_stack.ss\_size = sizeof(stack\_2);

// 协程2的任务函数return后，pthread线程将从协程1的yield调用的返回点处继续执行。

ctx[2].uc\_link = &ctx[1];

// 指定协程2的任务函数为func\_2。

makecontext(&ctx[2], func\_2, 0);

// 执行点1，将cpu当前各寄存器的值存入ctx[0]，将ctx[2]中存储的寄存器值加载到cpu寄存器中，

// 也就是main函数在这里yield，开始执行协程2的任务函数func\_2。

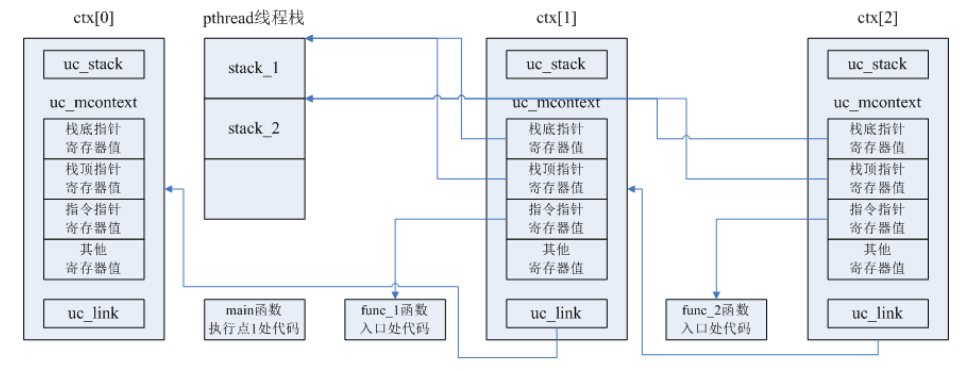
swapcontext(&ctx[0], &ctx[2]);

// 执行点5，main函数从这里resume恢复执行。

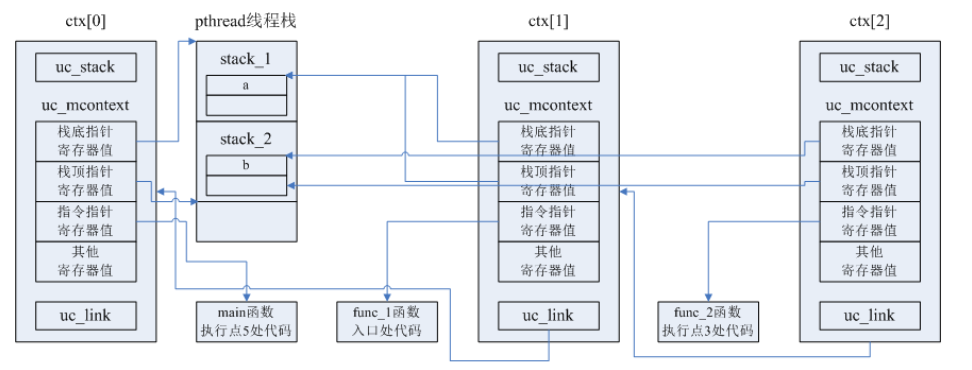
return 0;

}

在上述程序中，pthread系统线程执行到main函数的执行点1时，内存布局如下图所示，协程1和协程2的私有栈内存是main函数的局部变量，均分配在执行main函数的pthread的线程栈上。并且此时还未执行到swapcontext(&ctx[0], &ctx[2])，所以ctx[0]内的值都是空的：



pthread执行了main函数中的swapcontext(&ctx[0], &ctx[2])后，main函数（也可以认为是一个协程）yield，pthread开始执行协程2的任务函数func\_2，在func\_2中执行swapcontext(&ctx[2], &ctx[1])后，协程2 yield，pthread开始执行协程1的任务函数func\_1，pthread执行到func\_1内的执行点2时，内存布局如下图所示，此时main函数和协程2都已被挂起，ctx[0]存储了pthread线程栈的基底地址和栈顶地址，以及main函数执行点5处代码的地址，ctx[2]存储了stack\_2的基底地址和栈顶地址，以及func\_2函数执行点3处代码的地址，协程1正在被执行过程中，没有被挂起，所以ctx[1]相比之前没有变化：



pthread执行func\_1中的swapcontext(&ctx[1], &ctx[2])后，协程1被挂起，ctx[1]存储了stack\_1的基底地址和栈顶地址，以及func\_1函数执行点4处代码的地址，pthread转去执行协程2的任务函数的下一条代码，也就是协程2被resume，从func\_2函数的执行点3处恢复执行，接着func\_2就return了，由于ctx[2].uc\_link = &ctx[1]，pthread再次转去执行协程1的任务函数的下一条代码，协程1被resume，从func\_1函数的执行点4处恢复执行，再接着func\_1函数return，又由于ctx[1].uc\_link = &ctx[0]，pthread又去执行main函数的下一条代码，main函数被resume，从执行点5处恢复恢复执行，至此协程1和协程2都执行完毕，main函数也将要return了。这个过程可以称作main函数、协程1、协程2分别在一个pthread线程上被调度执行。

#### brpc的bthread任务定义

上面的协程示例程序可以认为是实现了N:1用户级线程，即所有协程都在一个系统线程pthread上被调度执行。**N:1协程的一个问题就是如果其中一个协程的任务函数在执行阻塞的网络I/O，或者在等待互斥锁，整个pthread系统线程就被挂起，其他的协程当然也无法得到执行了**。brpc在N:1协程的基础上做了扩展，实现了M:N用户级线程，即N个pthread系统线程去调度执行M个协程（M远远大于N），一个pthread有其私有的任务队列，队列中存储等待执行的若干协程，**一个pthread执行完任务队列中的所有协程后，也可以去其他pthread的任务队列中拿协程任务，即work-steal机制**，这样的话如果一个协程在执行较为耗时的操作时，同一任务队列中的其他协程有机会被调度到其他pthread上去执行，从而实现了全局的最大并发。并且brpc也实现了协程级的互斥与唤醒，即Butex机制，通过Butex，一个协程在等待网络I/O或等待互斥锁的时候，会被自动yield让出cpu，在适当时候会被其他协程唤醒，恢复执行。关于Butex的详情参见[这篇文章](https://github.com/ronaldo8210/brpc_source_code_analysis/blob/master/docs/butex.md)。

说明：

“work-steal”机制，也被称为“工作窃取”机制，是一种在并行计算或多线程环境中用于分配任务的策略。其核心思想是一个线程（或处理器）可以从其他线程（或处理器）的任务队列中“窃取”任务来执行，以充分利用计算资源并提高并行效率。

在Go语言的运行时系统中，这种机制被用于分配Goroutine到可用的线程（在Go中，这些线程被称为M，即Machine）上执行。当一个线程（M）完成了当前的Goroutine后，它会首先检查与它关联的本地队列。如果队列中还有Goroutine，就取出一个来执行。如果本地队列已经为空，线程会尝试从其他线程的队列中“窃取”一个Goroutine来执行。如果所有的队列都为空，线程就会进入一个全局的队列中等待，直到有新的Goroutine被创建。这种机制确保了所有的线程都能尽可能地保持忙碌，从而充分利用多核处理器的计算能力。

在更一般的上下文中，work-steal机制的实现通常涉及将大任务分割为若干个互不依赖的子任务，并将这些子任务分别放到不同的队列里。每个队列对应一个线程，线程和队列一一对应。当某个线程完成了自己队列中的所有任务时，它会尝试从其他线程的队列中窃取任务来执行。为了减少窃取任务线程和被窃取任务线程之间的竞争，通常会使用双端队列。被窃取任务线程永远从双端队列的头部拿任务执行，而窃取任务线程永远从双端队列的尾部拿任务执行。

这种机制的主要优点包括：

充分利用线程进行并行计算，减少线程间的竞争。

减少了线程间的同步开销，提高了整体性能。

总的来说，work-steal机制是一种有效的任务分配策略，特别适用于需要高度并行化和充分利用多核处理器资源的场景。

在brpc中一个协程任务可以称作一个bthread，一个bthread在内存中表示为一个TaskMeta对象，TaskMeta对象会被分配在ResourcePool中，TaskMeta类的主要的成员变量有：

* fn & arg：应用程序设置的bthread的任务处理函数及其参数。
* ContextualStack\* stack：ContextualStack结构的定义为：

struct ContextualStack {

// 缓存cpu寄存器上下文的结构，相当于posix的ucontext结构。

bthread\_fcontext\_t context;

StackType stacktype;

// bthread私有栈空间。

StackStorage storage;

};

* local\_storage：用于记录一些bthread运行状态（如各类统计值）等的一块内存。和ContextualStack::storage不能搞混。
* version\_butex：指向一个Butex对象头节点的指针。

### 多核环境下pthread调度执行bthread的过程

#### 调度执行bthread的主要数据结构

在一个线上环境系统中，会产生大量的bthread，系统的cpu核数有限，如何让大量的bthread在有限的cpu核心上得到充分调度执行，实现全局的最大并发主要是由TaskGroup对象、TaskControl对象实现的。

1、每一个TaskGroup对象是系统线程pthread的线程私有对象，它内部包含有任务队列，并控制pthread如何执行任务队列中的众多bthread任务。TaskGroup中主要的成员有：

\_remote\_rq：如果一个pthread 1想让pthread 2执行bthread 1，则pthread 1会将bthread 1的tid压入pthread 2的TaskGroup的\_remote\_rq队列中。

\_rq：pthread 1在执行从自己私有的TaskGroup中取出的bthread 1时，如果bthread 1执行过程中又创建了新的bthread 2，则bthread 1将bthread 2的tid压入pthread 1的TaskGroup的\_rq队列中。

\_main\_tid & \_main\_stack：一个pthread会在TaskGroup::run\_main\_task()中执行while()循环，不断获取并执行bthread任务，一个pthread的执行流不是永远在bthread中，比如等待任务时，pthread没有执行任何bthread，执行流就是直接在pthread上。可以将pthread在“等待bthread-获取到bthread-进入bthread执行任务函数之前”这个过程也抽象成一个bthread，称作一个pthread的“调度bthread”或者“主bthread”，它的tid和私有栈就是\_main\_tid和\_main\_stack。

\_cur\_meta：当前正在执行的bthread的TaskMeta对象的地址。

2、TaskControl对象是全局的单例对象，主要成员有：

\_pl：ParkingLot类型的数组。ParkingLot对象用于bthread任务的等待-通知。

\_workers：pthread线程标识符的数组，表示创建了多少个pthread worker线程，每个pthread worker线程应拥有一个线程私有的TaskGroup对象。

\_groups：TaskGroup对象指针的数组。

TaskControl和TaskGroup的内存关系如下图所示：



#### 一个pthread调度执行私有TaskGroup的任务队列中各个bthread的过程

一个pthread调度执行私有TaskGroup任务队列中的各个bthread，这些bthread是在pthread上串行执行的，彼此间不会有竞争。一个bthread的执行过程可能会有三种状态：

1、bthread的任务处理函数执行完成。一个bthread的任务函数结束后，该bthread需要负责查看TaskGroup的任务队列中是否还有bthread，如果有，则pthread执行流直接进入下一个bthread的任务函数中去执行；如果没有，则执行流返回pthread的调度bthread，等待其他pthread传递新的bthread；

2、bthread在任务函数执行过程中yield挂起，则pthread去执行任务队列中下一个bthread，如果任务队列为空，则执行流返回pthread的调度bthread，等待其他pthread传递新的bthread。挂起的bthread何时恢复运行取决于具体的业务场景，它应该被某个bthread唤醒，与pthread的调度无关。这样的例子有负责向TCP连接写数据的bthread因等待inode输出缓冲可写而被yield挂起、等待Butex互斥锁的bthread被yield挂起等。

3、bthread在任务函数执行过程中可以创建新的bthread，因为新的bthread一般是优先级更高的bthread，所以pthread执行流立即进入新bthread的任务函数，原先的bthread被重新加入到任务队列的尾部，不久后它仍然可以被pthread执行。但由于work-steal机制，它不一定会在原先的pthread执行，可能会被steal到其他pthread上执行。

按照以上的原则，分析下brpc中的实现过程。

1、TaskControl创建一个pthread worker线程和其私有的TaskGroup对象时，pthread在TaskGroup::run\_main\_task上开启无限循环：

void TaskGroup::run\_main\_task() {

bvar::PassiveStatus<double> cumulated\_cputime(

get\_cumulated\_cputime\_from\_this, this);

std::unique\_ptr<bvar::PerSecond<bvar::PassiveStatus<double> > > usage\_bvar;

TaskGroup\* dummy = this;

bthread\_t tid;

// 等待一个可执行的bthread，可能从\_rq中取得其他pthead压入的bthread id，

// 也可能从其他pthread worker线程的TaskGroup中steal一个bthread id。

while (wait\_task(&tid)) {

// 拿到一个bthread，执行流进入bthread的任务函数。

TaskGroup::sched\_to(&dummy, tid);

// run\_main\_task()恢复执行的开始执行点。

DCHECK\_EQ(this, dummy);

DCHECK\_EQ(\_cur\_meta->stack, \_main\_stack);

// 这里有些疑问，尚不确定何种情景下会执行下面这段代码。

if (\_cur\_meta->tid != \_main\_tid) {

TaskGroup::task\_runner(1/\*skip remained\*/);

}

if (FLAGS\_show\_per\_worker\_usage\_in\_vars && !usage\_bvar) {

char name[32];

#if defined(OS\_MACOSX)

snprintf(name, sizeof(name), "bthread\_worker\_usage\_%" PRIu64,

pthread\_numeric\_id());

#else

snprintf(name, sizeof(name), "bthread\_worker\_usage\_%ld",

(long)syscall(SYS\_gettid));

#endif

usage\_bvar.reset(new bvar::PerSecond<bvar::PassiveStatus<double> >

(name, &cumulated\_cputime, 1));

}

}

// stop\_main\_task() was called.

// Don't forget to add elapse of last wait\_task.

current\_task()->stat.cputime\_ns += butil::cpuwide\_time\_ns() - \_last\_run\_ns;

}

2、wait\_task()函数负责等待一个bthread，如果当前没有bthread可执行，则pthread会挂起。

3、TaskGroup::sched\_to(TaskGroup\*\* pg, bthread\_t next\_tid)的作用是根据将要执行的bthread的tid在O(1)时间内定位到bthread的TaskMeta对象的地址（TaskMeta是分配在ResourcePool中的，关于ResourcePool可以参考[这篇文章](https://github.com/ronaldo8210/brpc_source_code_analysis/blob/master/docs/resource_pool.md)），并确保bthread的私有栈空间已创建、context结构已分配，进而调用TaskGroup::sched\_to(TaskGroup\*\* pg, TaskMeta\* next\_meta)：

void TaskGroup::sched\_to(TaskGroup\*\* pg, TaskMeta\* next\_meta) {

TaskGroup\* g = \*pg;

#ifndef NDEBUG

if ((++g->\_sched\_recursive\_guard) > 1) {

LOG(FATAL) << "Recursively(" << g->\_sched\_recursive\_guard - 1

<< ") call sched\_to(" << g << ")";

}

#endif

// Save errno so that errno is bthread-specific.

const int saved\_errno = errno;

void\* saved\_unique\_user\_ptr = tls\_unique\_user\_ptr;

// 获取当前正在执行的bthread的TaskMeta对象的地址。

TaskMeta\* const cur\_meta = g->\_cur\_meta;

const int64\_t now = butil::cpuwide\_time\_ns();

const int64\_t elp\_ns = now - g->\_last\_run\_ns;

g->\_last\_run\_ns = now;

cur\_meta->stat.cputime\_ns += elp\_ns;

if (cur\_meta->tid != g->main\_tid()) {

// 如果一个bthread在执行过程中生成了新的bthread，会走到这里。

g->\_cumulated\_cputime\_ns += elp\_ns;

}

// 递增当前bthread的切换次数。

++cur\_meta->stat.nswitch;

// 递增worker线程pthread上的bthread切换次数。

++ g->\_nswitch;

// Switch to the task

if (\_\_builtin\_expect(next\_meta != cur\_meta, 1)) {

// 将\_cur\_meta指向下一个将要执行的bthread的TaskMeta对象的指针。

g->\_cur\_meta = next\_meta;

// Switch tls\_bls

// tls\_bls存储的是当前bthread的一些运行期数据（统计量等），执行切换动作前，将tls\_bls的内容复制到

// 当前bthread的私有storage空间中，再将tls\_bls重新指向将要执行的bthread的私有storage。

cur\_meta->local\_storage = tls\_bls;

tls\_bls = next\_meta->local\_storage;

// Logging must be done after switching the local storage, since the logging lib

// use bthread local storage internally, or will cause memory leak.

if ((cur\_meta->attr.flags & BTHREAD\_LOG\_CONTEXT\_SWITCH) ||

(next\_meta->attr.flags & BTHREAD\_LOG\_CONTEXT\_SWITCH)) {

LOG(INFO) << "Switch bthread: " << cur\_meta->tid << " -> "

<< next\_meta->tid;

}

if (cur\_meta->stack != NULL) {

if (next\_meta->stack != cur\_meta->stack) {

// 这里真正执行bthread的切换。

// 将执行pthread的cpu的寄存器的当前状态存入cur\_meta的context中，并将next\_meta的context中

// 的数据加载到cpu的寄存器中，开始执行next\_meta的任务函数。

jump\_stack(cur\_meta->stack, next\_meta->stack);

// 这里是cur\_meta代表的bthread的恢复执行点。

// bthread恢复执行的时候可能被steal到其他pthread上了，需要重置TaskGroup对象的指针g。

// probably went to another group, need to assign g again.

g = tls\_task\_group;

}

#ifndef NDEBUG

else {

// else pthread\_task is switching to another pthread\_task, sc

// can only equal when they're both \_main\_stack

CHECK(cur\_meta->stack == g->\_main\_stack);

}

#endif

}

// else because of ending\_sched(including pthread\_task->pthread\_task)

} else {

LOG(FATAL) << "bthread=" << g->current\_tid() << " sched\_to itself!";

}

while (g->\_last\_context\_remained) {

RemainedFn fn = g->\_last\_context\_remained;

g->\_last\_context\_remained = NULL;

fn(g->\_last\_context\_remained\_arg);

g = tls\_task\_group;

}

// Restore errno

errno = saved\_errno;

tls\_unique\_user\_ptr = saved\_unique\_user\_ptr;

#ifndef NDEBUG

--g->\_sched\_recursive\_guard;

#endif

\*pg = g;

}

4、一个bthread被执行时，pthread将执行TaskGroup::task\_runner()，在这个函数中会去执行TaskMeta对象的fn()，即应用程序设置的bthread任务函数。task\_runner()的关键代码如下：

void TaskGroup::task\_runner() {

TaskMeta\* const m = g->\_cur\_meta;

// 执行应用程序设置的任务函数，在任务函数中可能yield让出cpu，也可能产生新的bthread。

m->fn(m->arg);

// 任务函数执行完成后，需要唤起等待该任务函数执行结束的pthread/bthread。

butex\_wake\_except(m->version\_butex, 0);

// 将pthread线程执行流转入下一个可执行的bthread（普通bthread或pthread的调度bthread）。

ending\_sched(&g);

}

bthread任务函数结束完后会调用ending\_sched()，在ending\_sched()内会尝试从本地TaskGroup的任务队列中找出下一个bthread，或者从其他pthread的TaskGroup上steal一个bthread，如果没有bthread可用则下一个被执行的就是pthread的“调度bthread”，通过sched\_to()将pthread的执行流转入下一个bthread的任务函数。

5、一个bthread在自己的任务函数执行过程中想要挂起时，调用TaskGroup::yield(TaskGroup\*\* pg)，yield()内部会调用TaskGroup::sched(TaskGroup\*\* pg)，sched()也是负责将pthread的执行流转入下一个bthread（普通bthread或调度bthread）的任务函数。挂起的bthread在适当的时候会被其他bthread唤醒，即某个bthread会负责将挂起的bthread的tid重新加入TaskGroup的任务队列。

6、一个bthread 1在自己的任务函数执行过程中需要创建新的bthread 2时，会调用TaskGroup::start\_foreground()，在start\_foreground()内完成bthread 2的TaskMeta对象的创建，并调用sched\_to()让pthread去执行bthread 2的任务函数。pthread在真正执行bthread 2的任务函数前会将bthread 1的tid重新压入TaskGroup的任务队列，bthread 1不久之后会再次被调度执行。

### pthread线程间的Futex同步

#### spinlock和内核提供的同步机制存在的不足

在Futex出现之前，想要pthread系统线程等待一把锁，有两种实现方式：

1、使用spinlock自旋锁，实现起来也很简单：

std::atomic<int> flag;

void lock() {

int expect = 0;

while (!flag.compare\_exchange\_strong(expect, 1)) {

expect = 0;

}

}

void unlock() {

flag.store(0);

}

spinlock属于应用层的同步机制，直接运行在用户态，不涉及用户态-内核态的切换。spinlock存在的问题是，只适用于需要线程加锁的临界区代码段较小的场景，在这样的场景下可以认为一个线程加了锁后很快就会释放锁，等待锁的其他线程只需要很少的while()调用就可以得到锁；但如果临界区代码很长，一个线程加锁后会耗费相当一段时间去执行临界区代码，在这个线程释放锁之前其他线程只能不停地在while()中不断busy-loop，耗费了cpu资源，而且在应用程序层面又没有办法能让pthread系统线程挂起。

直接使用Linux提供的pthread\_mutex\_lock系统调用或者编程语言提供的操作互斥锁的API（如C++的std::unique\_lock），本质上是一样的，都是使用Linux内核提供的同步机制，可以让pthread系统线程挂起，让出CPU。但缺点是如果直接使用Linux内核提供的同步机制，每一次lock、unlock操作都是一次系统调用，需要进行用户态-内核态的切换，存在一定的性能开销，但lock、unlock的时候不一定会有线程间的竞争，在没有线程竞争的情况下没有必要进行用户态-内核态的切换。

#### Futex设计原理

Futex机制可以认为是结合了spinlock和内核态的pthread线程锁，它的设计意图是：

1、一个线程在加锁的时候，在用户态使用原子操作执行“尝试加锁，如果当前锁变量值为0，则将锁变量值更新为1，返回成功；如果当前锁变量值为1，说明之前已有线程拿到了锁，返回失败”这个动作，如果加锁成功，则可直接去执行临界区代码；如果加锁失败，则用类似pthread\_mutex\_lock这样的系统调用将当前线程挂起。由于可能有多个线程同时被挂起，所以必须将各个被挂起线程的信息存入一个与锁相关的等待队列中；

2、一个线程在释放锁的时候，也是用原子操作将锁变量的值改回0，并且如果与锁相关的等待队列不为空，则释放锁的线程必须使用内核提供的系统调用去唤醒因等待锁而被挂起的线程，具体是唤醒一个线程还是唤醒全部线程视使用场景而定；

3、由上可见，使用Futex机制，在没有线程竞争的情况下，在用户层就可以完成临界区代码的加锁解锁，只有在确实有线程竞争的情况下才会使用内核提供的系统调用实现线程的挂起与唤醒；

4、在实现Futex的时候有一个细节需要注意，Futex的代码如果像下面这样写：

void lock() {

while (!trylock()) {

wait(); // 使用内核提供的系统调用挂起当前线程。

}

}

这样的实现存在一个问题，在trylock()和wait()间存在一个时间窗口，在这个时间窗口中锁变量可能发生改变。比如一个线程A调用trylock()返回失败，在调用wait()前，锁被之前持有锁的线程B释放，线程A再调用wait()就会被永久挂起，永远不会再被唤醒了。因此需要在wait()内部再次判断锁变量是否仍为在trylock()内看到的旧值，如果不是，则wait()应直接返回，再次去执行trylock()。

#### brpc中Futex的实现

brpc实现了Futex机制，主要代码在src/bthread/sys\_futex.cpp中，SimuFutex类定义了一个锁的等待队列计数等统计量，另外有两个函数分别负责wait和wake：

* SimuFutex类：

class SimuFutex {

public:

SimuFutex() : counts(0)

, ref(0) {

pthread\_mutex\_init(&lock, NULL);

pthread\_cond\_init(&cond, NULL);

}

~SimuFutex() {

pthread\_mutex\_destroy(&lock);

pthread\_cond\_destroy(&cond);

}

public:

pthread\_mutex\_t lock;

pthread\_cond\_t cond;

// 有多少个线程在等待一个锁的时候被挂起。

int32\_t counts;

int32\_t ref;

};

* futex\_wait\_private函数：

// addr1是锁变量的地址，expected是在外层调用spinlock时看到的锁变量的值。

int futex\_wait\_private(void\* addr1, int expected, const timespec\* timeout) {

// 调用InitFutexMap初始化全局的std::unordered\_map<void\*, SimuFutex>\* 类型的s\_futex\_map，

// InitFutexMap仅被执行一次。

if (pthread\_once(&init\_futex\_map\_once, InitFutexMap) != 0) {

LOG(FATAL) << "Fail to pthread\_once";

exit(1);

}

std::unique\_lock<pthread\_mutex\_t> mu(s\_futex\_map\_mutex);

SimuFutex& simu\_futex = (\*s\_futex\_map)[addr1];

++simu\_futex.ref;

mu.unlock();

int rc = 0;

{

std::unique\_lock<pthread\_mutex\_t> mu1(simu\_futex.lock);

// 判断锁\*addr1的当前最新值是否等于expected期望值。

if (static\_cast<butil::atomic<int>\*>(addr1)->load() == expected) {

// 锁\*addr1的当前最新值与expected期望值相等，可以使用系统调用将当前线程挂起。

// 因为有一个线程为了等待锁而将要被挂起，锁\*addr1相关的counts计数器需要递增1。

++simu\_futex.counts;

// 调用pthread\_cond\_wait将当前线程挂起，并释放simu\_futex.lock锁。

if (timeout) {

timespec timeout\_abs = butil::timespec\_from\_now(\*timeout);

if ((rc = pthread\_cond\_timedwait(&simu\_futex.cond, &simu\_futex.lock, &timeout\_abs)) != 0) {

// pthread\_cond\_timedwait返回时会再次对simu\_futex.lock上锁。

errno = rc;

rc = -1;

}

} else {

if ((rc = pthread\_cond\_wait(&simu\_futex.cond, &simu\_futex.lock)) != 0) {

// pthread\_cond\_wait返回时会再次对simu\_futex.lock上锁。

errno = rc;

rc = -1;

}

}

// 当前线程已被唤醒并持有了锁\*addr1，counts计数器递减1。

--simu\_futex.counts;

} else {

// 锁\*addr1的当前最新值与expected期望值不等，需要再次执行上层的spinlock。

errno = EAGAIN;

rc = -1;

}

}

std::unique\_lock<pthread\_mutex\_t> mu1(s\_futex\_map\_mutex);

if (--simu\_futex.ref == 0) {

s\_futex\_map->erase(addr1);

}

mu1.unlock();

return rc;

}

* futex\_wake\_private函数：

int futex\_wake\_private(void\* addr1, int nwake) {

if (pthread\_once(&init\_futex\_map\_once, InitFutexMap) != 0) {

LOG(FATAL) << "Fail to pthread\_once";

exit(1);

}

std::unique\_lock<pthread\_mutex\_t> mu(s\_futex\_map\_mutex);

auto it = s\_futex\_map->find(addr1);

if (it == s\_futex\_map->end()) {

mu.unlock();

return 0;

}

SimuFutex& simu\_futex = it->second;

++simu\_futex.ref;

mu.unlock();

int nwakedup = 0;

int rc = 0;

{

std::unique\_lock<pthread\_mutex\_t> mu1(simu\_futex.lock);

nwake = (nwake < simu\_futex.counts)? nwake: simu\_futex.counts;

for (int i = 0; i < nwake; ++i) {

// 唤醒指定数量的在锁\*addr1上挂起的线程。

if ((rc = pthread\_cond\_signal(&simu\_futex.cond)) != 0) {

errno = rc;

break;

} else {

++nwakedup;

}

}

}

std::unique\_lock<pthread\_mutex\_t> mu2(s\_futex\_map\_mutex);

if (--simu\_futex.ref == 0) {

s\_futex\_map->erase(addr1);

}

mu2.unlock();

return nwakedup;

}

### Butex机制：bthread粒度的挂起与唤醒

#### bthread粒度挂起与唤醒的设计原理

由于bthread任务是在pthread系统线程中执行，在需要bthread间互斥的场景下不能使用pthread级别的锁（如pthread\_mutex\_lock或者C++的unique\_lock等），否则pthread会被挂起，不仅当前的bthread中止执行，pthread私有的TaskGroup的任务队列中其他bthread也无法在该pthread上调度执行。因此需要在应用层实现bthread粒度的互斥机制，一个bthread被挂起时，pthread仍然要保持运行状态，保证TaskGroup任务队列中的其他bthread的正常执行不受影响。

要实现bthread粒度的互斥，方案如下：

1、在同一个pthread上执行的多个bthread是串行执行的，不需要考虑互斥；

2、如果位于heap内存上或static静态区上的一个对象A可能会被在不同pthread执行的多个bthread同时访问，则为对象A维护一个互斥锁（一般是一个原子变量）和等待队列，同时访问对象A的多个bthread首先要竞争锁，假设三个bthread 1、2、3分别在pthread 1、2、3上执行，bthread 1、bthread 2、bthread 3同时访问heap内存上的一个对象A，这时就产生了竞态，假设bthread 1获取到锁，可以去访问对象A，bthread 2、bthread 3先将自身必要的信息（bthread的tid等）存入等待队列，然后自动yiled，让出cpu，让pthread 2、pthread 3继续去执行各自私有TaskGroup的任务队列中的下一个bthread，这就实现了bthread粒度的挂起；

3、bthread 1访问完对象A后，通过查询对象A的互斥锁的等待队列，能够得知bthread 2、bthread 3因等待锁而被挂起，bthread 1负责将bthread 2、3的tid重新压入某个pthread（不一定是之前执行执行bthread 2、3的pthread 2、3）的TaskGroup的任务队列，bthread 2、3就能够再次被pthread执行，这就实现了bthread粒度的唤醒。

下面分析下brpc是如何实现bthread粒度的挂起与唤醒的。

#### brpc中Butex的源码解释

brpc实现bthread互斥的主要代码在src/bthread/butex.cpp中：

1、首先解释下Butex、ButexBthreadWaiter等主要的数据结构：

struct BAIDU\_CACHELINE\_ALIGNMENT Butex {

Butex() {}

~Butex() {}

// 锁变量的值。

butil::atomic<int> value;

// 等待队列，存储等待互斥锁的各个bthread的信息。

ButexWaiterList waiters;

internal::FastPthreadMutex waiter\_lock;

};

// 等待队列实际上是个侵入式双向链表，增减元素的操作都可在O(1)时间内完成。

typedef butil::LinkedList<ButexWaiter> ButexWaiterList;

// ButexWaiter是LinkNode的子类，LinkNode里只定义了指向前后节点的指针。

struct ButexWaiter : public butil::LinkNode<ButexWaiter> {

// tids of pthreads are 0

// tid就是64位的bthread id。

// Butex实现了bthread间的挂起&唤醒，也实现了bthread和pthread间的挂起&唤醒，

// 一个pthread在需要的时候可以挂起，等待适当的时候被一个bthread唤醒，线程挂起不需要tid，填0即可。

// pthread被bthread唤醒的例子可参考brpc的example目录下的一些client.cpp示例程序，执行main函数的pthread

// 会被挂起，某个bthread执行完自己的任务后会去唤醒pthread。

bthread\_t tid;

// Erasing node from middle of LinkedList is thread-unsafe, we need

// to hold its container's lock.

butil::atomic<Butex\*> container;

};

// bthread需要挂起时，会在栈上创建一个ButexBthreadWaiter对象（对象存储在bthread的私有栈空间内）并加入等待队列。

struct ButexBthreadWaiter : public ButexWaiter {

// 执行bthread的TaskMeta结构的指针。

TaskMeta\* task\_meta;

TimerThread::TaskId sleep\_id;

// 状态标记，根据锁变量当前状态是否发生改变，waiter\_state会被设为不同的值。

WaiterState waiter\_state;

// expected\_value存储的是当bthread竞争互斥锁失败时锁变量的值，由于从bthread竞争互斥锁失败到bthread挂起

// 有一定的时间间隔，在这个时间间隔内锁变量的值可能会发生变化，也许锁已经被释放了，那么之前竞争锁失败的bthread

// 就不应挂起，否则可能永远不会被唤醒了，它应该放弃挂起动作，再去竞争互斥锁。所以一个bthread在执行挂起动作前

// 一定要再次去查看锁变量的当前最新值，只有锁变量当前最新值等于expected\_value时才能真正执行挂起动作。

int expected\_value;

Butex\* initial\_butex;

// 指向全局唯一的TaskControl单例对象的指针。

TaskControl\* control;

};

// 如果是pthread挂起，则创建ButexPthreadWaiter对象并加入等待队列。

struct ButexPthreadWaiter : public ButexWaiter {

butil::atomic<int> sig;

};



以上面的bthread 1获得了互斥锁、bthread 2和bthread 3因等待互斥锁而被挂起的场景为例，Butex的内存布局如下图所示，展现了主要的对象间的内存关系，注意ButexBthreadWaiter变量是分配在bthread的私有栈上的：

2、执行bthread挂起的函数是butex\_wait：

// arg是指向Butex::value锁变量的指针，expected\_value是bthread竞争锁失败时锁变量的值。

int butex\_wait(void\* arg, int expected\_value, const timespec\* abstime) {

// 通过arg定位到Butex对象的地址。

Butex\* b = container\_of(static\_cast<butil::atomic<int>\*>(arg), Butex, value);

// 如果锁变量当前最新值不等于expected\_value，则锁的状态发生了变化，当前bthread不再执行挂起动作，

// 直接返回，在外层代码中继续去竞争锁。

if (b->value.load(butil::memory\_order\_relaxed) != expected\_value) {

errno = EWOULDBLOCK;

// Sometimes we may take actions immediately after unmatched butex,

// this fence makes sure that we see changes before changing butex.

butil::atomic\_thread\_fence(butil::memory\_order\_acquire);

return -1;

}

TaskGroup\* g = tls\_task\_group;

if (NULL == g || g->is\_current\_pthread\_task()) {

// 当前代码不在bthread中执行而是在直接在pthread上执行，调用butex\_wait\_from\_pthread让pthread挂起。

return butex\_wait\_from\_pthread(g, b, expected\_value, abstime);

}

// 创建ButexBthreadWaiter类型的局部变量bbw，bbw是分配在bthread的私有栈空间上的。

ButexBthreadWaiter bbw;

// tid is 0 iff the thread is non-bthread

bbw.tid = g->current\_tid();

bbw.container.store(NULL, butil::memory\_order\_relaxed);

bbw.task\_meta = g->current\_task();

bbw.sleep\_id = 0;

bbw.waiter\_state = WAITER\_STATE\_READY;

bbw.expected\_value = expected\_value;

bbw.initial\_butex = b;

bbw.control = g->control();

if (abstime != NULL) {

// Schedule timer before queueing. If the timer is triggered before

// queueing, cancel queueing. This is a kind of optimistic locking.

if (butil::timespec\_to\_microseconds(\*abstime) <

(butil::gettimeofday\_us() + MIN\_SLEEP\_US)) {

// Already timed out.

errno = ETIMEDOUT;

return -1;

}

bbw.sleep\_id = get\_global\_timer\_thread()->schedule(

erase\_from\_butex\_and\_wakeup, &bbw, \*abstime);

if (!bbw.sleep\_id) { // TimerThread stopped.

errno = ESTOP;

return -1;

}

}

#ifdef SHOW\_BTHREAD\_BUTEX\_WAITER\_COUNT\_IN\_VARS

bvar::Adder<int64\_t>& num\_waiters = butex\_waiter\_count();

num\_waiters << 1;

#endif

// release fence matches with acquire fence in interrupt\_and\_consume\_waiters

// in task\_group.cpp to guarantee visibility of `interrupted'.

bbw.task\_meta->current\_waiter.store(&bbw, butil::memory\_order\_release);

// pthread在执行任务队列中下一个bthread前，会先执行wait\_for\_butex()将刚创建的bbw对象放入锁的等待队列。

g->set\_remained(wait\_for\_butex, &bbw);

// 当前bthread yield让出cpu，pthread会从TaskGroup的任务队列中取出下一个bthread去执行。

TaskGroup::sched(&g);

// 这里是butex\_wait()恢复执行时的开始执行点。

// erase\_from\_butex\_and\_wakeup (called by TimerThread) is possibly still

// running and using bbw. The chance is small, just spin until it's done.

BT\_LOOP\_WHEN(unsleep\_if\_necessary(&bbw, get\_global\_timer\_thread()) < 0,

30/\*nops before sched\_yield\*/);

// If current\_waiter is NULL, TaskGroup::interrupt() is running and using bbw.

// Spin until current\_waiter != NULL.

BT\_LOOP\_WHEN(bbw.task\_meta->current\_waiter.exchange(

NULL, butil::memory\_order\_acquire) == NULL,

30/\*nops before sched\_yield\*/);

#ifdef SHOW\_BTHREAD\_BUTEX\_WAITER\_COUNT\_IN\_VARS

num\_waiters << -1;

#endif

bool is\_interrupted = false;

if (bbw.task\_meta->interrupted) {

// Race with set and may consume multiple interruptions, which are OK.

bbw.task\_meta->interrupted = false;

is\_interrupted = true;

}

// If timed out as well as value unmatched, return ETIMEDOUT.

if (WAITER\_STATE\_TIMEDOUT == bbw.waiter\_state) {

errno = ETIMEDOUT;

return -1;

} else if (WAITER\_STATE\_UNMATCHEDVALUE == bbw.waiter\_state) {

errno = EWOULDBLOCK;

return -1;

} else if (is\_interrupted) {

errno = EINTR;

return -1;

}

return 0;

}

static void wait\_for\_butex(void\* arg) {

ButexBthreadWaiter\* const bw = static\_cast<ButexBthreadWaiter\*>(arg);

Butex\* const b = bw->initial\_butex;

{

BAIDU\_SCOPED\_LOCK(b->waiter\_lock);

// 再次判断锁变量的当前最新值是否与expected\_value相等。

if (b->value.load(butil::memory\_order\_relaxed) != bw->expected\_value) {

// 锁变量的状态发生了变化，bw代表的bthread不能挂起，要去重新竞争锁。

// 但bw代表的bthread之前已经yield让出cpu了，所以下面要将bw代表的bthread的id再次放入TaskGroup的

// 任务队列，让它恢复执行。

// 将bw的waiter\_state更改为WAITER\_STATE\_UNMATCHEDVALUE，表示锁的状态已发生改变。

bw->waiter\_state = WAITER\_STATE\_UNMATCHEDVALUE;

} else if (bw->waiter\_state == WAITER\_STATE\_READY/\*1\*/ &&

!bw->task\_meta->interrupted) {

// 将bw加入到锁的等待队列，这才真正完成bthread的挂起，然后直接返回。

b->waiters.Append(bw);

bw->container.store(b, butil::memory\_order\_relaxed);

return;

}

}

// 锁状态发生变化的情况下，才执行后面的代码。

// b->container is NULL which makes erase\_from\_butex\_and\_wakeup() and

// TaskGroup::interrupt() no-op, there's no race between following code and

// the two functions. The on-stack ButexBthreadWaiter is safe to use and

// bw->waiter\_state will not change again.

unsleep\_if\_necessary(bw, get\_global\_timer\_thread());

// 将bw代表的bthread的tid重新加入TaskGroup的任务队列。

tls\_task\_group->ready\_to\_run(bw->tid);

// FIXME: jump back to original thread is buggy.

// // Value unmatched or waiter is already woken up by TimerThread, jump

// // back to original bthread.

// TaskGroup\* g = tls\_task\_group;

// ReadyToRunArgs args = { g->current\_tid(), false };

// g->set\_remained(TaskGroup::ready\_to\_run\_in\_worker, &args);

// // 2: Don't run remained because we're already in a remained function

// // otherwise stack may overflow.

// TaskGroup::sched\_to(&g, bw->tid, false/\*2\*/);

}

3、执行bthread唤醒的函数有butex\_wake（唤醒正在等待一个互斥锁的一个bthread）、butex\_wake\_all（唤醒正在等待一个互斥锁的所有bthread）、butex\_wake\_except（唤醒正在等待一个互斥锁的除了指定bthread外的其他bthread），下面解释butex\_wake的源码：

// arg是Butex::value锁变量的地址。

int butex\_wake(void\* arg) {

// 通过arg定位到Butex对象的地址。

Butex\* b = container\_of(static\_cast<butil::atomic<int>\*>(arg), Butex, value);

ButexWaiter\* front = NULL;

{

BAIDU\_SCOPED\_LOCK(b->waiter\_lock);

// 如果锁的等待队列为空，直接返回。

if (b->waiters.empty()) {

return 0;

}

// 取出锁的等待队列中第一个ButexWaiter对象的指针，并将该ButexWaiter对象从等待队列中移除。

front = b->waiters.head()->value();

front->RemoveFromList();

front->container.store(NULL, butil::memory\_order\_relaxed);

}

if (front->tid == 0) {

// ButexWaiter对象的tid=0说明挂起的是pthread系统线程，调用内核提供的系统调用将pthread线程唤醒。

wakeup\_pthread(static\_cast<ButexPthreadWaiter\*>(front));

return 1;

}

ButexBthreadWaiter\* bbw = static\_cast<ButexBthreadWaiter\*>(front);

unsleep\_if\_necessary(bbw, get\_global\_timer\_thread());

// 将挂起的bthread的tid压入TaskGroup的任务队列，实现了将挂起的bthread唤醒。

TaskGroup\* g = tls\_task\_group;

if (g) {

TaskGroup::exchange(&g, bbw->tid);

} else {

bbw->control->choose\_one\_group()->ready\_to\_run\_remote(bbw->tid);

}

return 1;

}

## Client端执行流程

参考：

<https://zhuanlan.zhihu.com/p/101032908>

### 概述

### 原理

#### Channel

Client指发起请求的一端，在brpc中没有对应的实体，取而代之的是brpc::Channel，它代表和一台或一组服务器的交互通道，Client和Channel在角色上的差别在实践中并不重要，你可以把Channel视作Client。

Channel可以被所有线程共用，你不需要为每个线程创建独立的Channel，也不需要用锁互斥。不过Channel的创建和Init并不是线程安全的，请确保在Init成功后再被多线程访问，在没有线程访问后再析构。

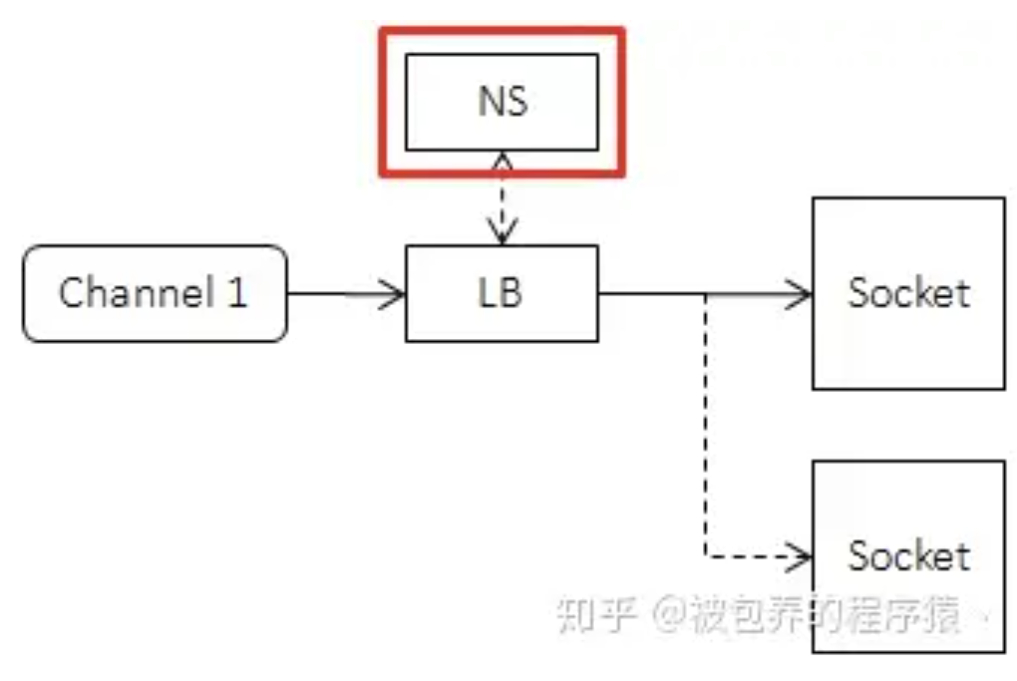
一些RPC实现中有ClientManager的概念，包含了Client端的配置信息和资源管理。brpc不需要这些，以往在ClientManager中配置的线程数、长短连接等等要么被加入了brpc::ChannelOptions，要么可以通过gflags全局配置，这么做的好处：方便、共用资源、生命周期（析构ClientManager由框架完成不容易出错)。 就像大部分类那样，Channel必须在Init之后才能使用，options为NULL时所有参数取默认值。 Init函数分为连接一台服务器和连接服务集群。

#### 命名服务

参考：

命名服务Naming Service：<https://zhuanlan.zhihu.com/p/101073888>

命名服务把一个名字映射为可修改的机器列表，在client端的位置如下：



命名服务

有了命名服务后client记录的是一个名字，而不是每一台下游机器。而当下游机器变化时，就只需要修改命名服务中的列表，而不需要逐台修改每个上游。这个过程也常被称为“解耦上下游”。当然在具体实现上，上游会记录每一台下游机器，并定期向命名服务请求或被推送最新的列表，以避免在RPC请求时才去访问命名服务。使用命名服务一般不会对访问性能造成影响，对命名服务的压力也很小。

#### 命名服务过滤器

#### 负载均衡

参考：

bRPC负载均衡之Round Robin：<https://zhuanlan.zhihu.com/p/101560608>

bRPC负载均衡之DoublyBufferedData：<https://zhuanlan.zhihu.com/p/101338245>

<https://github.com/apache/brpc/blob/master/docs/cn/load_balancing.md>

##### rr/Round Robin

##### wrr/Weighted Round Robin

##### la/locality-aware

##### c\_murmurhash or c\_md5

##### 从集群宕机后恢复时的客户端限流

参考：<https://zhuanlan.zhihu.com/p/101853641>

#### 发起访问

一般来说，我们不直接调用Channel.CallMethod，而是通过protobuf生成的XXX\_Stub，过程更像是“调用函数”。stub内没什么成员变量，建议在栈上创建和使用，而不必new，当然你也可以把stub存下来复用Channel::CallMethod和stub访问都是线程安全的，可以被所有线程同时访问。比如：

XXX\_Stub stub(&channel);

stub.some\_method(controller, request, response, done);

或者

XXX\_Stub(&channel).some\_method(controller, request, response, done);

一个例外是http/h2 client。访问http服务和protobuf没什么关系，直接调用CallMethod即可，除了Controller和done均为NULL。

##### 同步访问

指的是：CallMethod会阻塞到收到server端返回response或发生错误（包括超时）。

同步访问中的response/controller不会在CallMethod后被框架使用，它们都可以分配在栈上。注意，如果request/response字段特别多字节数特别大的话，还是更适合分配在堆上。

MyRequest request;

MyResponse response;

brpc::Controller cntl;

XXX\_Stub stub(&channel);

request.set\_foo(...);

cntl.set\_timeout\_ms(...);

stub.some\_method(&cntl, &request, &response, NULL);

if (cntl->Failed()) {

// RPC失败了. response里的值是未定义的，勿用。

} else {

// RPC成功了，response里有我们想要的回复数据。

}

##### 异步访问

指的是：给CallMethod传递一个额外的回调对象done，CallMethod在发出request后就结束了，而不是在RPC结束后。当server端返回response或发生错误（包括超时）时，done->Run()会被调用。对RPC的后续处理应该写在done->Run()里，而不是CallMethod后。

由于CallMethod结束不意味着RPC结束，response/controller仍可能被框架及done->Run()使用，它们一般得创建在堆上，并在done->Run()中删除。如果提前删除了它们，那当done->Run()被调用时，将访问到无效内存。

你可以独立地创建这些对象，并使用NewCallback生成done，也可以把Response和Controller作为done的成员变量，一起new出来，一般使用前一种方法。

发起异步请求后Request和Channel也可以立刻析构。这两样和response/controller是不同的。注意:这是说Channel的析构可以立刻发生在CallMethod之后，并不是说析构可以和CallMethod同时发生，删除正被另一个线程使用的Channel是未定义行为（很可能crash）。

###### 使用NewCallback

static void OnRPCDone(MyResponse\* response, brpc::Controller\* cntl) {

// unique\_ptr会帮助我们在return时自动删掉response/cntl，防止忘记。gcc 3.4下的unique\_ptr是模拟版本。

std::unique\_ptr<MyResponse> response\_guard(response);

std::unique\_ptr<brpc::Controller> cntl\_guard(cntl);

if (cntl->Failed()) {

// RPC失败了. response里的值是未定义的，勿用。

} else {

// RPC成功了，response里有我们想要的数据。开始RPC的后续处理。

}

// NewCallback产生的Closure会在Run结束后删除自己，不用我们做。

}

MyResponse\* response = new MyResponse;

brpc::Controller\* cntl = new brpc::Controller;

MyService\_Stub stub(&channel);

MyRequest request; // 你不用new request,即使在异步访问中.

request.set\_foo(...);

cntl->set\_timeout\_ms(...);

stub.some\_method(cntl, &request, response, google::protobuf::NewCallback(OnRPCDone, response, cntl));

###### 继承Google：：protobuf：：closure

使用NewCallback的缺点是要分配三次内存：response, controller, done。如果profiler证明这儿的内存分配有瓶颈，可以考虑自己继承Closure，把response/controller作为成员变量，这样可以把三次new合并为一次。但缺点就是代码不够美观，如果内存分配不是瓶颈，别用这种方法。

class OnRPCDone: public google::protobuf::Closure {

public:

void Run() {

// unique\_ptr会帮助我们在return时自动delete this，防止忘记。gcc 3.4下的unique\_ptr是模拟版本。

std::unique\_ptr<OnRPCDone> self\_guard(this);

if (cntl->Failed()) {

// RPC失败了. response里的值是未定义的，勿用。

} else {

// RPC成功了，response里有我们想要的数据。开始RPC的后续处理。

}

}

MyResponse response;

brpc::Controller cntl;

}

OnRPCDone\* done = new OnRPCDone;

MyService\_Stub stub(&channel);

MyRequest request; // 你不用new request,即使在异步访问中.

request.set\_foo(...);

done->cntl.set\_timeout\_ms(...);

stub.some\_method(&done->cntl, &request, &done->response, done);

### 执行流程

#### 无异常状态下的一次完整RPC请求过程

##### 数据发送过程涉及到的主要数据结构

Channel对象：表示客户端与一台服务器或一组服务器的连接通道。

Controller对象：存储一次完整的RPC请求的Context以及各种状态。

Butex对象：实现bthread粒度的互斥锁，管理锁的挂起等待队列。

Id对象：同步一次RPC过程中的各个bthread（发送数据、处理服务器的响应、处理超时均由不同的bthread完成）。

##### 数据发送过程中的系统内存布局与多线程执行状态

以brpc自带的实例程序example/multi\_threaded\_echo\_c++/client.cpp为例，结合Client端内存布局的变化过程和多线程执行过程，阐述无异常状态下（所有发送数据都及时得到响应，没有超时、没有因服务器异常等原因引发的请求重试）的Client发送请求直到处理响应的过程。

该程序运行后，会与单台服务器建立一条TCP长连接，创建thread\_num个bthread（后续假设thread\_num=3）在此TCP连接上发送、接收数据，不涉及连接池、负载均衡。RPC使用同步方式，这里的同步是指bthread间的同步：负责发送数据的bthread完成发送操作后，不能结束，而是需要挂起，等待负责接收服务器响应的bthread将其唤醒后，再恢复执行。挂起时会让出cpu，pthread可继续执行任务队列中的其他bthread。

具体运行过程为：

1、在main函数的栈上创建一个Channel对象，并初始化Channel的协议类型、连接类型、RPC超时时间、请求重试次数等参数，上述参数后续会被赋给所有通过此Channel的RPC请求；

2、在main函数中调用bthread\_start\_background创建3个bthread，此时TaskControl、TaskGroup对象都并不存在，所以此时需要在heap内存上创建它们（惰性初始化方式，不是在程序启动时就创建所有的对象，而是到对象确实要被用到时才去创建）：

一个TaskControl单例对象；

N个TaskGroup对象（后续假设N=4），每个TaskGroup对应一个系统线程pthread，是pthread的线程私有对象，每个pthread启动后以自己的TaskGroup对象的run\_main\_task函数作为主工作函数，在该函数内执行无限循环，不断地从TaskGroup的任务队列中取得bthread id、通过id找到bthread对象、去执行bthread任务函数。

3、在TaskMeta对象池中创建3个TaskMeta对象（每个TaskMeta等同一个bthread），每个TaskMeta的fn函数指针指向client.cpp中定义的static类型函数sender，sender就是bthread的任务处理函数。每个TaskMeta创建完后，按照散列规则找到一个TaskGroup对象，并将tid（也就是bthread的唯一标识id）压入该TaskGroup对象的\_remote\_rq队列中（TaskGroup所属的pthread线程称为worker线程，worker线程自己产生的bthread的tid会被压入自己私有的TaskGroup对象的\_rq队列，本实例中的main函数所在线程不属于worker线程，所以main函数所在的线程生成的bthread的tid会被压入找到的TaskGroup对象的\_rq队列）；

4、main函数执行到这里，不能直接结束（否则Channel对象会被马上析构，所有RPC无法进行），必须等待3个bthread全部执行sender函数结束后，main才能结束。

main函数所在线程挂起的实现机制是，将main函数所在线程的信息存储在ButexPthreadWaiter中，并加入到TaskMeta对象的version\_butex指针所指的Butex对象的等待队列waiters中，TaskMeta的任务函数fn执行结束后，会从waiters中查找到“之前因等待TaskMeta的任务函数fn执行结束而被挂起的”pthread线程，再将其唤醒。关于Butex机制的细节，可参见[这篇文章](https://github.com/ronaldo8210/brpc_source_code_analysis/blob/master/docs/butex.md)；

main函数所在的系统线程在join bthread 1的时候就被挂起，等待在wait\_pthread函数处。bthread 1执行sender函数结束后，唤醒main函数的线程，main函数继续向下执行，去join bthread 2。如果此时bthread 2仍在运行，则再将存储了main函数所在线程信息的一个新的ButexPthreadWaiter加入到bthread 2对应的TaskMeta对象的version\_butex指针所指的Butex对象的等待队列waiters中，等到bthread 2执行完sender函数后再将main函数所在线程唤醒。也可能当main函数join bthread 2的时候bthread 2已经运行完成，则join操作直接返回，接着再去join bthread 3；

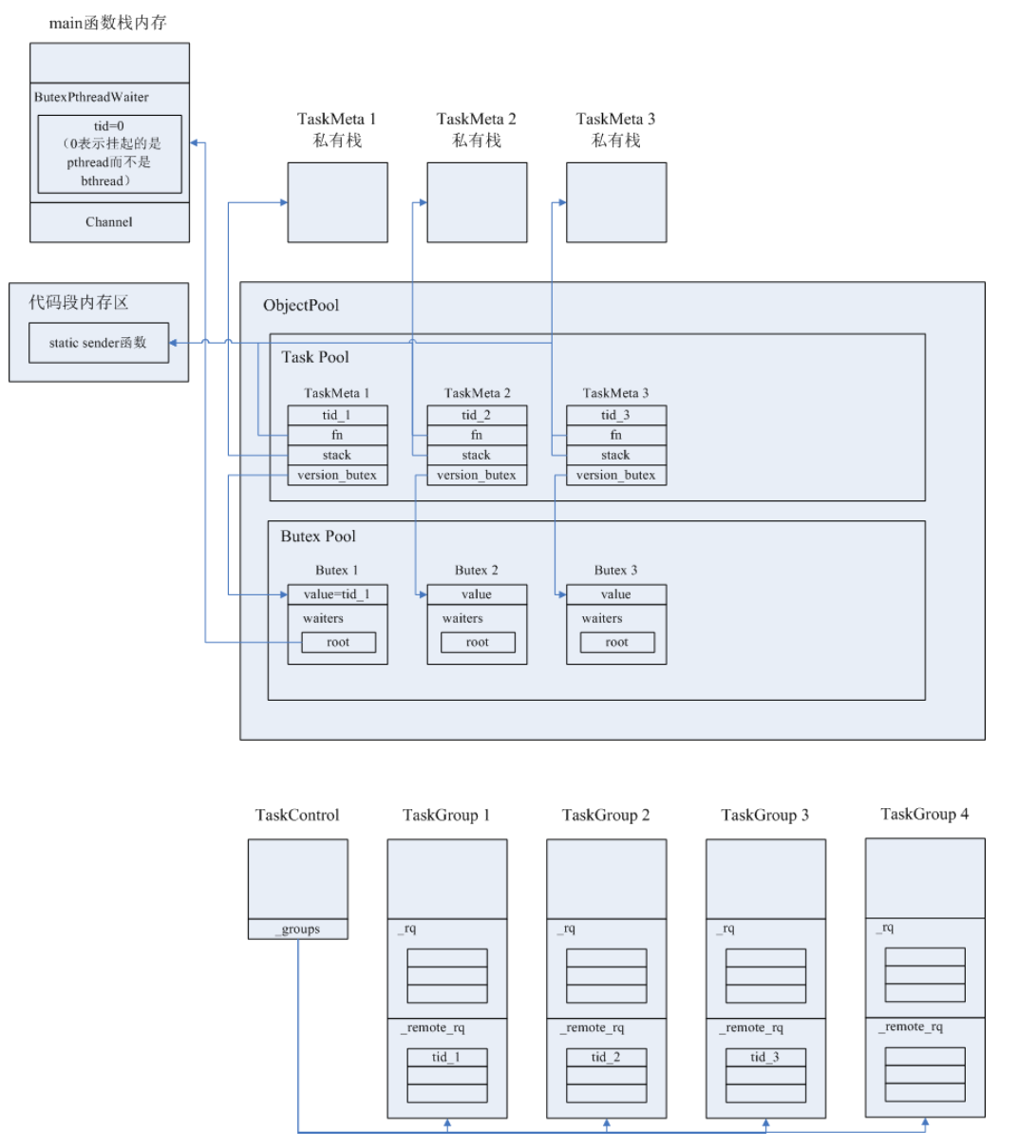
只有当三步join操作全部返回后，main函数才结束。

5、此时Client进程内部的线程状态是：

bthread状态：三个bthread 1、2、3已经创建完毕，各自的bthread id已经被分别压入不同的TaskGroup对象（设为TaskGroup 1、2、3）的任务队列\_remote\_rq中；

pthread状态：此时进程中存在5个pthread线程：3个pthread即将从各自私有的TaskGroup对象的\_remote\_rq中拿到bthread id，将要执行bthread id对应的TaskMeta对象的任务函数；1个pthread仍然阻塞在run\_main\_task函数上，等待新任务到来通知；main函数所在线程被挂起，等待bthread 1执行结束。

6、此时Client进程内部的内存布局如下图所示，由于bthread 1、2、3还未开始运行，未分配任何局部变量，所以此时各自的私有栈都是空的：



7、TaskGroup 1、2、3分别对应的3个pthread开始执行各自拿到的bthread的任务函数，即client.cpp中的static类型的sender函数。由于各个bthread有各自的私有栈空间，所以sender中的局部变量request、response、Controller对象均被分配在bthread的私有栈内存上；

8、根据protobuf的标准编程模式，3个执行sender函数的bthread都会执行Channel的CallMethod函数，CallMethod负责的工作为：

CallMethod函数的入参为各个bthread私有的request、response、Controller，CallMethod内部会为Controller对象的相关成员变量赋值，包括RPC起始时间戳、最大重试次数、RPC超时时间、Backup Request超时时间、标识一次RPC过程的唯一id correlation\_id等等。Controller对象可以认为是存储了一次RPC过程的所有Context上下文信息;

在CallMethod函数中不存在线程间的竞态，CallMethod本身是线程安全的。而Channel对象是main函数的栈上对象，main函数所在线程已被挂起，直到3个bthread全部执行完成后才会结束，所以Channel对象的生命期贯穿于整个RPC过程;

构造Controller对象相关联的Id对象，Id对象的作用是同步一次RPC过程中的各个bthread，因为在一次RPC过程中，发送请求、接收响应、超时处理均是由不同的bthread负责，各个bthread可能运行在不同的pthread上，因此这一次RPC过程的Controller对象可能被上述不同的bthread同时访问，也就是相当于被不同的pthread并发访问，产生竞态。此时不能直接让某个pthread去等待线程锁，那样会让pthread挂起，阻塞该pthread私有的TaskGroup对象的任务队列中其他bthread的执行。因此如果一个bthread正在访问Controller对象，此时位于不同pthread上的其他bthread若想访问Controller，必须将自己的bthread信息加入到一个等待队列中，yield让出cpu，让pthread继续去执行任务队列中下一个bthread。正在访问Controller的bthread让出访问权后，会从等待队列中找到挂起的bthread，并将其bthread id再次压入某个TaskGroup的任务队列，这样就可让原先为了等待Controller访问权而挂起的bthread得以从yield点恢复执行。这就是bthread级别的挂起-唤醒的基本原理，这样也保证所有pthread是wait-free的。

在CallMethod中会通过将Id对象的butex指针指向的Butex结构的value值置为“locked\_ver”表示Id对象已被锁，即当前发送数据的bthread正在访问Controller对象。在本文中假设发送数据后正常接收到响应，不涉及重试、RPC超时等，所以不深入阐述Id对象，关于Id的细节请参考[这篇文章](https://github.com/ronaldo8210/brpc_source_code_analysis/blob/master/docs/client_bthread_sync.md)。

9、pthread线程执行流程接着进入Controller的IssueRPC函数，在该函数中：

按照指定协议格式将RPC过程的首次请求的call\_id、RPC方法名、实际待发送数据打包成报文；

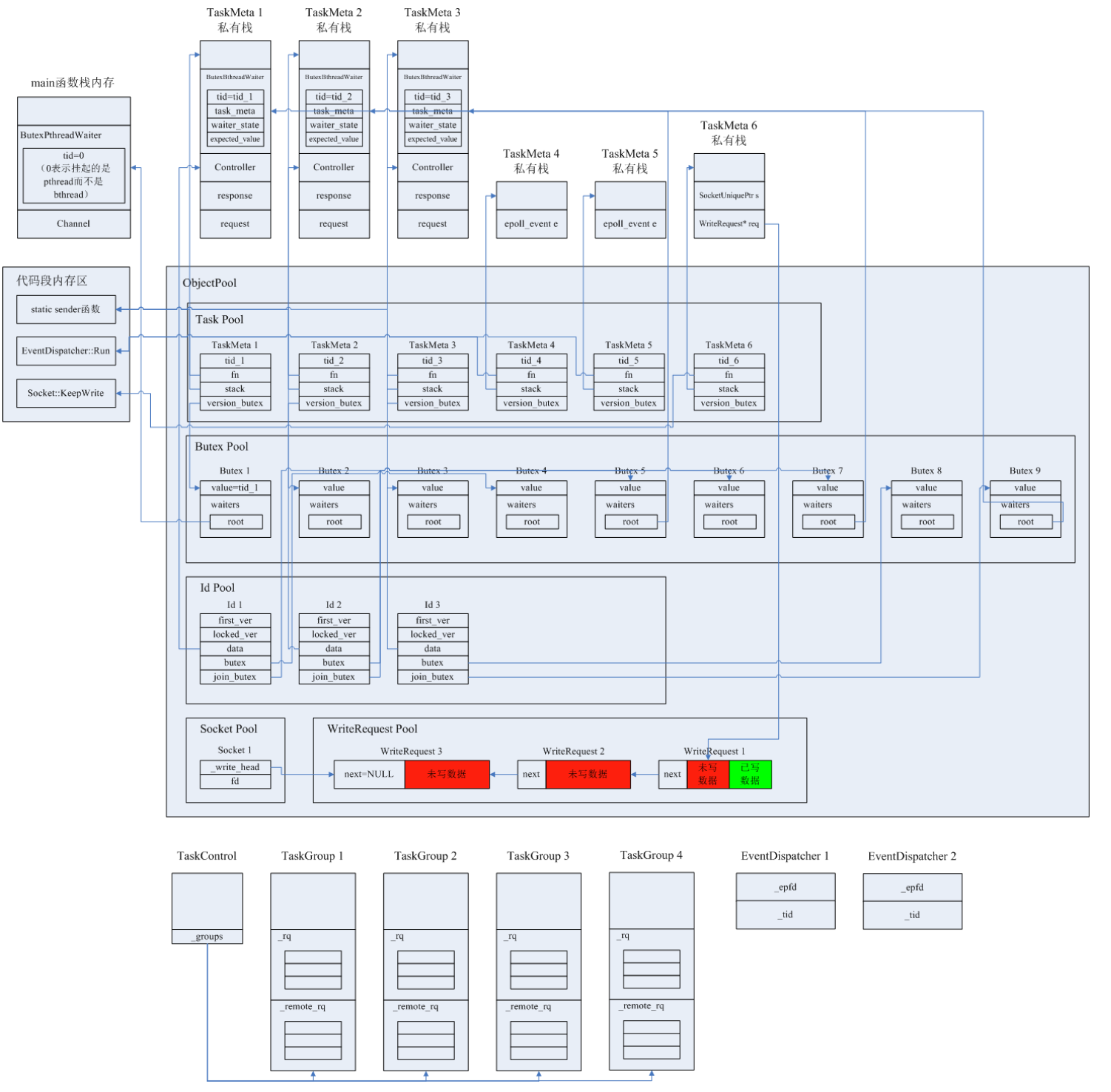
调用Socket::Write函数执行实际的发送数据过程。Socket对象表示Client与单台Server的连接。向fd写入数据的细节过程参考[这篇文章](https://github.com/ronaldo8210/brpc_source_code_analysis/blob/master/docs/io_write.md)；

在实际发送数据前需要先建立与Server的TCP长连接，并惰性初始化event\_dispatcher\_num个EventDispatcher对象（假设event\_dispatcher\_num=2），从而新建2个bthread 4和5，并将它们的任务处理函数设为static类型的EventDispatcher::RunThis函数，当bthread 4、5得到pthread执行时，会调用epoll\_wait检测是否有I/O事件触发。brpc是没有专门的I/O pthread线程的；

从Socket::Write函数返回后，调用bthread\_id\_unlock释放对Controller对象的独占访问。

10、因为RPC使用synchronous同步方式，所以bthread完成数据发送后调用bthread\_id\_join将自身挂起，让出cpu，等待负责接收服务器响应的bthread来唤醒。此时Client进程内部的线程状态是：bthread 1、2、3都已挂起，执行bthread任务的pthread 1、2、3分别跳出了bthread 1、2、3的任务函数，回到TaskGroup::run\_main\_task函数继续等待新的bthread任务，因为在向fd写数据的过程中通常会新建一个KeepWrite bthread（bthread 6），假设这个bthread的id被压入到TaskGroup 4的任务队列中，被pthread 4执行，所以pthread 1、2、3此时没有新bthread可供执行，处于系统线程挂起状态。

11、此时Client进程内部的内存布局如下图所示，注意各个类型对象分配在不同的内存区，比如Butex对象、Id对象分配在heap上，Controller对象、ButexBthreadWaiter对象分配在bthread的私有栈上：



12、KeepWrite bthread完成工作后，3个请求都被发出，假设服务器正常返回了3个响应，由于3个响应是在一个TCP连接上接收的，所以bthread 4、5二者只会有一个通过epoll\_wait()检测到fd可读，并新建一个bthread 7去负责将fd的inode输入缓存中的数据读取到应用层，在拆包过程中，解析出一条Response，就为这个Response的处理再新建一个bthread，目的是实现响应读取+处理的最大并发。因此Response 1在bthread 8中被处理，Response 2在bthread 9中被处理，Response 3在bthread 7中被处理（最后一条Response不需要再新建bthread了，直接在bthread 7本地处理即可）。bthread 8、9、7会将Response 1、2、3分别复制到相应Controller对象的response中，这时应用程序就会看到响应数据了。bthread 8、9、7也会将挂起的bthread 1、2、3唤醒，bthread 1、2、3会恢复执行，可以对Controller对象中的response做一些操作，并开始发送下一个RPC请求。

#### RPC请求可能遇到的多种异常及应对策略

在一次RPC过程中（假设一个客户端fd与一个服务端fd原先已正常建立TCP长连接），客户端向服务器发送请求时可能会遇到如下各类异常：

1、服务器限流

服务器接收请求的队列的长度超出阈值，此时服务器不能再负责处理新到的请求，否则服务器可能压力过大而崩溃。

2、服务器正在退出

服务器正在退出服务，此时服务器进程虽然存活，但不能响应客户端的请求。

3、服务器进程正常关闭fd

服务器进程正常关闭fd包括三种情况：

服务器进程主动调用close关闭fd；

服务器进程主动调用exit；

在shell下kill或ctrl+c杀掉服务器进程。

以上三种情况都会正常关闭服务端fd，TCP连接上会将FIN发给客户端，客户端的epoll会通知客户端的fd可读，触发的事件是EPOLLIN和EPOLLRDHUP，客户端应用层对fd执行read调用的返回值是0。

4、机器断电导致的服务器进程异常关闭

服务器进程所在的机器可能在如下时刻断电：

* 客户端发送数据过程中服务器机器断电

这里的客户端发送数据指的是客户端机器内核TCP/IP协议栈向服务器发送数据，客户端应用层进程只是将数据写入fd的内核inode发送缓存，操作系统内核的TCP/IP协议栈负责将inode发送缓存里的数据真正发送到网络上。如果服务器机器断电而客户端没有任何动作，则客户端不会被告知对端异常，只有当客户端的TCP/IP协议栈执行发送数据的动作时，由于滑动窗口无法滑动、无法收到已发数据的Ack等原因，协议栈会感知到对端异常，epoll会触发EPOLLERR事件通知应用层TCP连接已异常断开（此处的逻辑需要进一步查阅Linux内核协议栈代码来证实）。

* 客户端已将数据发到网络上，服务器进程接到请求数据前机器断电

客户端在超时时间内无法接收到响应，走超时处理的逻辑。

* 服务器进程处理请求过程中断电

同上，客户端会判断超时。

5、网络断线

断线包括网线断开、客户端与服务器间的路由器故障等，发生的场景有：

* 客户端发送数据时网络断线

与发送数据时服务器断电的逻辑一致，内核在发送数据时检测到TCP连接异常，epoll触发EPOLLERR事件通知应用层TCP连接异常（需进一步查阅Linux内核协议栈代码证实）。

* 客户端发送请求成功，请求数据在网络上传输时断线，服务器未收到请求

客户端在超时时间内无法收到响应，走超时处理的逻辑。服务器内核协议栈没有读写动作，不会得知此时TCP连接异常。

* 客户端发送请求成功，服务器处理请求过程中网络断线

客户端在超时时间内无法收到响应，走超时处理的逻辑。服务器内核协议栈发送响应数据过程中检测到TCP连接异常。

* 客户端发送请求成功，服务器处理请求后发送响应成功，响应数据在网络上传输时断线，客户端未收到响应

客户端走超时处理的逻辑。服务器内核协议栈接下来没有读写动作，不会得知此时TCP连接异常。

* 由于断线，服务器可能只接收到请求数据的一个片段，或者客户端只接收到响应数据的一个片段

按照非法数据处理。

综上，可以总结出RPC客户端需要面对的异常状况有：

服务器虽存活但不能处理请求

超时时间内没有接收到响应

服务器被正常关闭

因服务器机器断电或网络断线导致的TCP连接异常

下面阐述brpc是如何处理发送RPC请求过程中可能出现的上述4种异常的。

##### RPC请求发送过程可能会遇到的异常情况

服务器虽存活但不能处理请求

1、如果服务器由于软件层面的限制不能处理请求，会发回给客户端一个响应，响应包含错误码，服务器接收队列过大的错误码为ELIMIT，服务器正在退出的错误码为ELOGOFF，服务器已停止服务的错误码为EHOSTDOWN，客户端收到这几类错误码后，如果是通过连接池与多台服务器连接，会重新将请求发往一台新的服务器，这就是一次重试。如果服务器返给客户端的带有错误码的response由于TCP连接异常等原因丢失，则只能按请求超时的逻辑处理。

重试的请求也可能遇到服务器不能处理的情况，会产生下一次重试，重试不能无限产生，必须限定一次RPC过程的最大重试次数，只有下面三个条件同时满足时，才允许进行重试：

未到RPC超时时间

有剩余的重试次数

服务器返回的错误码允许重试，或者TCP连接出现异常时也可以重试

2、超时时间内没有接收到响应

brpc规定每一次RPC都有一个超时时间，超时时间是在Channel对象中设置，通过此Channel的所有RPC的超时时间是一样的，填充在每个RPC对应的Controller对象中。在超时时间内可以进行若干次重试发送，但只要过了超时时间还没收到想要的响应，RPC就会结束，就算重试产生的响应在超时时间过后马上到来也会被忽略。

3、服务器被正常关闭

客户端对一个fd的read操作返回0后，该fd不再可用，关闭fd，客户端应将后续的请求发给连接池中的其他TCP连接。如果fd上有bthread因等待服务器的Response而被挂起，则这些bthread需要被唤醒，唤醒后去执行RPC重试的逻辑。

4、因服务器机器断电或网络断线导致的TCP连接异常

当一个TCP连接出现异常后，epoll会在对应的fd上触发EPOLLERR事件，brpc会立即将fd所属的Socket对象置为不可用，如果之前已经有bthread通过此Socket对象发送了数据、正在等待服务器Response而被挂起，则这些bthread需要被唤醒，唤醒后去执行RPC重试的逻辑。

##### brpc对各类发送异常的处理方式

#### 重试&Backup Request

#### 同一RPC过程中各个bthread间的互斥

##### 一次RPC过程中需要bthread互斥的场景

在一次RPC过程中，由于设置RPC超时定时器和开启Backup Request机制，不同的bthread可能会同时操作本次RPC独有的Controller结构，会存在下列几种竞态情况：

1、第一次Request发出后，在backup\_request\_ms内未收到响应，触发Backup Request定时器，试图发送Backup Request的同时可能收到了第一次Request的Response，发送Backup Request的bthread和处理Response的bthread需要做互斥；

2、假设没有开启Backup Request机制，处理Response（可能是第一次Request的Response，也可能是某一次重试的Response）时刚好到了RPC超时时间，处理Response的bthread和处理RPC超时定时任务的bthread需要做互斥；

3、第一次Request或者任何一次重试Request，与Backup Request可能同时刻收到Response，分别处理两个Response的bthread间需要做互斥；

4、第一次Request或者任何一次重试Request，与Backup Request可能同时刻收到Response，此时也可能到了RPC超时时间，分别处理两个Response的bthread和处理RPC超时定时任务的bthread，三者之间需要做互斥。

##### bthread互斥过程涉及到的数据结构

一次RPC过程中，会有一个Controller对象贯穿本次RPC的始终，Controller对象内存储本次RPC的各类数据和各种状态，bthread间的互斥就是指同一时刻只能有一个bthread在访问Controller对象。互斥主要是通过Id对象和Butex对象实现的。

1、Id对象

brpc会在每一次RPC过程开始阶段创建本次RPC唯一的一个Id对象，用来保护Controller对象，互斥试图 同时访问Controller对象的多个bthread。Id对象主要成员有：

* first\_ver & locked\_ver

如果Id对象的butex锁变量的值（butex指针指向的Butex对象的value值）为first\_ver，表示Controller对象此时没有bthread在访问。此时如果有一个bthread试图访问Controller对象，则它可以取得访问权，先将butex锁变量的值置为locked\_ver后，再去访问Controller对象。

在locked\_ver的基础上又有contended\_ver、unlockable\_ver、end\_ver，contended\_ver = locked\_ver + 1，unlockable\_ver = locked\_ver + 2，end\_ver = locked\_ver + 3。

contended\_ver表示同时访问Controller对象的多个bthread产生了竞态。如果有一个bthread（bthread 1）在访问Controller对象结束后，观察到butex锁变量的值仍为locked\_ver，表示没有其他的bthread在等待访问Controller对象，bthread 1在将butex锁变量的值改为first\_ver后不会再有其他动作。如果在bthread 1访问Controller对象期间又有bthread 2试图访问Controller对象，bthread 2会观察到butex锁变量的值为locked\_ver，bthread 2首先将butex锁变量的值改为contended\_ver，意图就是告诉bthread 1产生了竞态。接着bthread 2需要将自身的bthread id等信息存储在butex锁变量的waiters等待队列中，yield让出cpu，让bthread 2所在的pthread去执行TaskGroup任务队列中的下一个bthread任务。当bthread 1完成对Controller对象的访问时，会观察到butex锁变量的值已被改为contended\_ver，bthread 1会到waiters队列中找到被挂起的bthread 2的id，通知TaskControl将bthread 2的id压入某一个TaskGroup的任务队列，这就是唤醒了bthread 2。bthread 1唤醒bthread 2后会将butex锁变量的值再次改回first\_ver。bthread 2重新被某一个pthread调度执行时，如果这期间没有其他bthread在访问Controller对象，bthread 2会观察到butex锁变量的值为first\_ver，这时bthread 2获得了Controller对象的访问权。

unlockable\_ver表示RPC即将完成，不允许再有bthread去访问Controller对象了。

end\_ver表示一次RPC结束后，Id对象被回收到对象池中后\*butex 和 \*join\_butex的值（butex和join\_butex都指向Butex对象的第一个元素value，value是个整型值，所以butex和join\_butex可转化为指向整型的指针）。end\_ver也是被回收的Id对象再次被分配给某一次RPC过程时的first\_ver值。

locked\_ver的取值和一次RPC的重试次数有关，locked\_ver = first\_ver + 重试次数 + 2，例如，如果一次RPC过程开始时分配给本次RPC的Id对象的first\_ver=1，本次RPC最多允许重试3次，则locked\_ver=6，本次RPC的唯一id \_correlation\_id=1，第一次请求的call\_id=2，第一次重试的call\_id=3，第二次重试的call\_id=4，第三次重试的call\_id=5，contended\_ver=7，unlockable\_ver=8，end\_ver=9（\_correlation\_id和call\_id的值只是举例，实际上\_correlation\_id和call\_id都是64bit整型值，前32bit为Id对象在对象池中的slot位移，后32bit为上述的1、2、3...等版本号。服务器返回的Response会回传call\_id，通过call\_id的前32bit可以在O(1)时间内定位到本次RPC对应的Id对象，方便进行后续的bthread互斥等操作）。

* mutex

类似futex的线程锁，由于试图同时访问同一Controller对象的若干bthread可能在不同的系统线程pthread上被执行，所以bthread在修改Id对象中的字段前需要先做pthread间的互斥。

* data

指向本次RPC唯一的一个Controller对象的指针。

* butex

指向一个Butex对象头节点的指针，该Butex对象用来互斥同时访问Controller对象的多个bthread，waiter队列存储被挂起的等待Controller对象访问权的bthread的tid等信息。

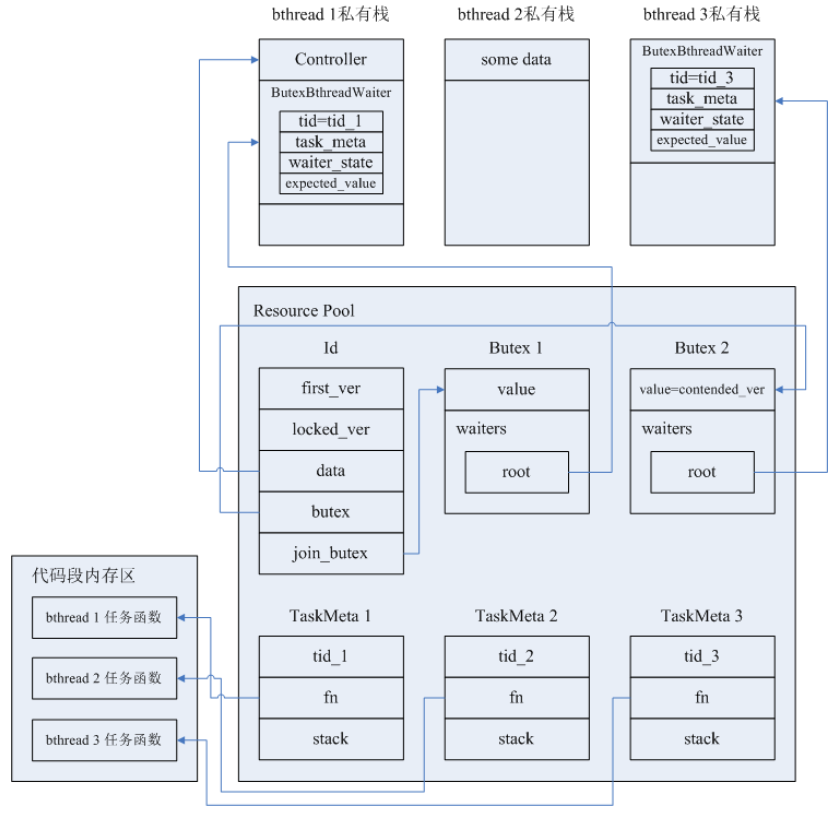
* join\_butex

指向另一个Butex对象头节点的指针，如果RPC过程是同步方式，该Butex对象用来同步发起RPC过程的bthread和成功将服务器的Response存入Controller对象的bthread：发起RPC过程的bthread（bthread 1）会执行第一次发送请求的动作，然后会将自己的bthread id等信息存储在join\_butex锁的waiters队列中，yield让出cpu，等待某一个能够成功将服务器的Response存入Controller对象的bthread将其唤醒。不论是处理第一次请求的Response的bthread，还是处理某一次重试请求的Response的bthread，只要有一个bthread将Response成功存入Controller对象，就从join\_butex锁的waiters队列中拿到bthread 1的bthread id，通知TaskControl将bthread 1的id压入某一个TaskGroup的任务队列，这就是唤醒了bthread 1，bthread 1被某一个pthread重新执行后会从Controller对象中得到Response，进行后续的Response处理工作。

2、Butex对象

Butex对象主要成员是ButexWaiterList类型的waiters，waiters是个等待队列（waiters实际上是一个侵入式链表，增删操作都会在O(1)时间内完成），等待Controller访问权的bthread会在私有栈上创建一个ButexBthreadWaiter对象，并加入到waiters中，ButexBthreadWaiter对象中包含挂起的bthread的tid等信息，释放Controller访问权的bthread可以从waiters队列中拿到挂起的bthread的tid，并负责将挂起的bthread的tid加入某个TaskGroup的任务队列，让它重新得到某个pthread的调度。

假设现在有bthread 1使用同步方式发起了一次RPC请求，发送请求后bthread 1被挂起，等待有bthread向Controller对象填充请求的Response，或者超时。一段时间后处理服务器Response的bthread 2和处理超时的bthread 3同时执行，bthread 2抢到Controller的访问权，bthread 3被挂起。此时Controller、Id、Butex几种对象间的内存关系如下图所示，注意Controller对象分配在bthread 1的私有栈上，两个ButexBthreadWaiter对象也分配在相应bthread的私有栈上，Id对象和Butex对象都是通过ResourcePool机制分配的，被分配在heap堆上，Butex 2的value值是contended\_ver，因为bthread 2访问Controller期间有bthread 3在排队等待，bthread 2释放Controller访问权后必须负责唤醒bthread 3，并且bthread 2成功向Controller写入了服务器的Response，满足bthread 1的唤醒条件，bthread 2还必须负责唤醒bthread 1。



##### brpc实现bthread互斥的源码解释

brpc实现bthread互斥的主要结构为Id和Butex，关于Butex的细节请见[这篇文章](https://github.com/ronaldo8210/brpc_source_code_analysis/blob/master/docs/butex.md)，Id相关的代码在src/bthread/id.cpp中，主要的一些函数如下：

* bthread\_id\_lock\_and\_reset\_range\_verbose：访问共享数据前，竞争butex锁、等待butex锁：

// bthread访问Controller对象前必须要执行bthread\_id\_lock，实际上是调用bthread\_id\_lock\_and\_reset\_range\_verbose。

// 在这个函数中bthread会根据锁变量（Id的butex指针指向的Butex结构中value）的当前值，来判断下一步的动作：

// 1、如果锁变量当前值=first\_ver，说明当前没有bthread在访问Controller，则把锁变量的值置为locked\_ver，

// 告诉后来的bthread“我正在访问Controller，其他bthread先等待”，再去访问Controller；

// 2、如果锁变量当前值=locked\_ver或contended\_ver，则当前bthread需要挂起，正在访问Controller的bthread结束访问后

// 会负责唤醒挂起的bthread。

// 参数中，id是请求的call\_id（要和RPC的correlation\_id区分开），\*pdata是共享对象（如Controller）的地址，

// range=RPC重试次数+2。

int bthread\_id\_lock\_and\_reset\_range\_verbose(

bthread\_id\_t id, void \*\*pdata, int range, const char \*location) {

// 通过id的前32bits，在O(1)时间内定位到Id对象的地址。

bthread::Id\* const meta = address\_resource(bthread::get\_slot(id));

if (!meta) {

return EINVAL;

}

// id\_ver是call\_id（一次RPC由于重试等因素可能产生多次call，每个call有其唯一id）。

const uint32\_t id\_ver = bthread::get\_version(id);

// butex指针指向的是Butex结构的第一个元素：整型变量value，这就是锁变量。

uint32\_t\* butex = meta->butex;

// 函数的局部变量都是分配在各个bthread的私有栈上的，所以每个bthread看到的不是同一个ever\_contended。

bool ever\_contended = false;

// 这段代码可以被位于不同pthread上的多个bthread同时执行，所以需要先加线程锁。

meta->mutex.lock();

while (meta->has\_version(id\_ver)) {

if (\*butex == meta->first\_ver) {

// 执行到这里，表示当前没有其他bthread在访问Controller。

// contended locker always wakes up the butex at unlock.

meta->lock\_location = location;

if (range == 0) {

// fast path

} else if (range < 0 ||

range > bthread::ID\_MAX\_RANGE ||

range + meta->first\_ver <= meta->locked\_ver) {

LOG\_IF(FATAL, range < 0) << "range must be positive, actually "

<< range;

LOG\_IF(FATAL, range > bthread::ID\_MAX\_RANGE)

<< "max range is " << bthread::ID\_MAX\_RANGE

<< ", actually " << range;

} else {

// range的值是“一次RPC的重试次数+2”，

// 如果first\_ver=1，一次RPC在超时时间内允许重试3次，则locked\_ver=6。

meta->locked\_ver = meta->first\_ver + range;

}

// 1、如果是第一个访问Controller的bthread走到这里，则把锁变量的值置为locked\_ver；

// 2、如果是曾经因等待锁而被挂起的bthread走到这里，则把锁变量的值置为contended\_ver。

\*butex = (ever\_contended ? meta->contended\_ver() : meta->locked\_ver);

// 锁变量已经被重置，后来的bthread看到锁变量最新值后就会得知已经有一个bthread在访问Controller，

// 当前bthread可以释放pthread线程锁了。

meta->mutex.unlock();

if (pdata) {

// 找到Controller对象的指针并返回。

\*pdata = meta->data;

}

return 0;

} else if (\*butex != meta->unlockable\_ver()) {

// 1、一个bthread（假设bthread id为C）执行到这里，锁变量的当前值（Butex的value值）

// 要么是locked\_ver，要么是contented\_ver：

// a、如果锁变量的当前值=locked\_ver，表示当前有一个bthread A正在访问Controller且还没有访问完成，

// 且锁的等待队列中没有其他bthread被挂起；

// b、如果锁变量的当前值=contented\_ver，表示当前不仅有一个bthread A正在访问Controller且还没有

// 访问完成，而且还有一个或多个bthread（B、D、E...）被挂起，等待唤醒。

// 2、执行到这段代码的bthread必须要挂起，挂起前先将锁变量的值置为contended\_ver，告诉正在访问Controller

// 的bthread，访问完Controller后，要负责唤醒挂起的bthread；

// 3、挂起是指：bthread将cpu寄存器的上下文存入context结构，让出cpu，执行这个bthread的pthread从TaskGroup

// 的任务队列中取出下一个bthread去执行。

// 将锁变量的值置为contended\_ver。

\*butex = meta->contended\_ver();

// 记住竞争锁失败时的锁变量的当前值，在bthread真正执行挂起动作前，要再次检查锁变量的最新值，只有挂起前的

// 锁变量最新值与expected\_ver相等，bthread才能真正挂起；如果不等，锁可能已被释放，bthread不能挂起，否则

// 可能永远无法被唤醒，这时bthread应该放弃挂起动作，再次去竞争butex锁。

uint32\_t expected\_ver = \*butex;

// 关键字段的重置已完成，可以释放pthread线程锁了。

meta->mutex.unlock();

// 已经出现了bthread间的竞态。

ever\_contended = true;

// 在butex\_wait内部，新建ButexWaiter结构保存该bthread的主要信息并将ButexWaiter加入锁的等待队列waiters

// 链表，然后yield让出cpu。

// bthread真正挂起前，要再次判断锁变量的最新值是否与expected\_ver相等。

if (bthread::butex\_wait(butex, expected\_ver, NULL) < 0 &&

errno != EWOULDBLOCK && errno != EINTR) {

return errno;

}

// 这里是bthread被唤醒后，恢复执行点。

// 之前挂起的bthread被重新执行，先要再次去竞争pthread线程锁。不一定能竞争成功，所以上层要有一个while循环

// 不断的去判断被唤醒的bthread抢到pthread线程锁后可能观察到的butex锁变量的各种不同值。

meta->mutex.lock();

} else { // bthread\_id\_about\_to\_destroy was called.

// Butex的value被其他bthread置为unlockable\_ver，Id结构将被释放回资源池，Controller结构将被析构，

// 即一次RPC已经完成，因此执行到这里的bthread直接返回，不会再有后续的动作。

meta->mutex.unlock();

return EPERM;

}

}

meta->mutex.unlock();

return EINVAL;

}

* bthread\_id\_unlock：释放butex锁，从锁的等待队列中唤醒一个bthread：

int bthread\_id\_unlock(bthread\_id\_t id) {

// 通过id的前32bits，在O(1)时间内定位到Id对象的地址。

bthread::Id\* const meta = address\_resource(bthread::get\_slot(id));

if (!meta) {

return EINVAL;

}

uint32\_t\* butex = meta->butex;

// Release fence makes sure all changes made before signal visible to

// woken-up waiters.

const uint32\_t id\_ver = bthread::get\_version(id);

// 竞争pthread线程锁。

meta->mutex.lock();

if (!meta->has\_version(id\_ver)) {

// call\_id非法，严重错误。

meta->mutex.unlock();

LOG(FATAL) << "Invalid bthread\_id=" << id.value;

return EINVAL;

}

if (\*butex == meta->first\_ver) {

// 一个bthread执行到这里，观察到的butex锁变量的当前值要么是locked\_ver，要么是contented\_ver，

// 不可能是first\_ver，否则严重错误。

meta->mutex.unlock();

LOG(FATAL) << "bthread\_id=" << id.value << " is not locked!";

return EPERM;

}

bthread::PendingError front;

if (meta->pending\_q.pop(&front)) {

meta->lock\_location = front.location;

meta->mutex.unlock();

//

if (meta->on\_error) {

return meta->on\_error(front.id, meta->data, front.error\_code);

} else {

return meta->on\_error2(front.id, meta->data, front.error\_code,

front.error\_text);

}

} else {

// 如果锁变量的当前值为contended\_ver，则有N（N>=1）个bthread挂在锁的waiters队列中，等待唤醒。

const bool contended = (\*butex == meta->contended\_ver());

// 重置锁变量的值为first\_ver，表示当前的bthread对Controller的独占性访问已完成，后续被唤醒的bthread可以去

// 竞争对Controller的访问权了。

\*butex = meta->first\_ver;

// 关键字段已完成更新，释放线程锁。

meta->mutex.unlock();

if (contended) {

// We may wake up already-reused id, but that's OK.

// 唤醒waiters等待队列中的一个bthread。

bthread::butex\_wake(butex);

}

return 0;

}

}

* bthread\_id\_join：负责RPC发送过程的bthread完成发送动作后，会调用bthread\_id\_join将自身挂起，等待处理服务器Response的bthread来唤醒。这时是挂在join\_butex锁的等待队列中的，与butex锁无关。

int bthread\_id\_join(bthread\_id\_t id) {

// 通过id的前32bits，在O(1)时间内定位到Id对象的地址。

const bthread::IdResourceId slot = bthread::get\_slot(id);

bthread::Id\* const meta = address\_resource(slot);

if (!meta) {

// The id is not created yet, this join is definitely wrong.

return EINVAL;

}

const uint32\_t id\_ver = bthread::get\_version(id);

uint32\_t\* join\_butex = meta->join\_butex;

while (1) {

meta->mutex.lock();

const bool has\_ver = meta->has\_version(id\_ver);

const uint32\_t expected\_ver = \*join\_butex;

meta->mutex.unlock();

if (!has\_ver) {

break;

}

// 将ButexWaiter挂在join\_butex锁的等待队列中，bthread yield让出cpu。

// bthread恢复执行的时候，一般是RPC过程已经完成时。

if (bthread::butex\_wait(join\_butex, expected\_ver, NULL) < 0 &&

errno != EWOULDBLOCK && errno != EINTR) {

return errno;

}

}

return 0;

}

### Client设置

### 使用

TDSQL3.0中，客户端使用场景如下：

在SQLEngine、MC、TDStore中均定义client，获取RPC的相关耗时信息，然后借助bvar::LatencyRecorder存储起来。

bvar::LatencyRecorder g\_latency\_recorder("client");

在MC中使用如下：

static void \*mc\_get(void \*args) {

// Normally, you should not call a Channel directly, but instead construct

// a stub Service wrapping it. stub can be shared by all threads as well.

MCGetParam \*mp = (MCGetParam \*)args;

metarpc::MetaService\_Stub \*stub =

new metarpc::MetaService\_Stub((static\_cast<google::protobuf::RpcChannel \*>(mp->channel)));

// pid\_t pid = getpid();

// const int number = pid \* 1000000 + mp->number;

while (!brpc::IsAskedToQuit()) {

brpc::Controller cntl;

metarpc::GetTxnTimestampRequest request;

metarpc::GetTxnTimestampResponse response;

// request.set\_engine\_id(number);

stub->GetTxnTimestamp(&cntl, &request, &response, NULL);

if (!cntl.Failed()) {

g\_latency\_recorder << cntl.latency\_us();

} else {

g\_error\_count << 1;

CHECK(brpc::IsAskedToQuit() || !FLAGS\_dont\_fail)

<< "error=" << cntl.ErrorText() << " latency=" << cntl.latency\_us();

if (FLAGS\_sleep\_time >= 100000) {

bthread\_usleep(FLAGS\_sleep\_time);

} else {

bthread\_usleep(50000);

}

}

}

free(stub);

return NULL;

}

从Region获取key信息：

static void \*get\_region\_from\_key(void \*args) {

MCGetParam \*mp = (MCGetParam \*)args;

metarpc::MetaService\_Stub \*stub =

new metarpc::MetaService\_Stub(static\_cast<google::protobuf::RpcChannel \*>(mp->channel));

pid\_t pid = getpid();

const std::string number = std::to\_string(pid \* 1000000 + mp->number);

while (!brpc::IsAskedToQuit()) {

sleep(1);

brpc::Controller cntl;

metarpc::GetRegionByKeyRequest request;

metarpc::GetRegionByKeyResponse response;

request.set\_data\_space\_type(metarpc::DATA\_SPACE\_TYPE\_SYSTEM);

std::string key = "\270" + std::to\_string(GetIndex());

LOG(INFO) << "key:" << key;

request.set\_key(key);

stub->GetRegionByKey(&cntl, &request, &response, NULL);

if (!cntl.Failed()) {

g\_latency\_recorder << cntl.latency\_us();

LOG(DEBUG) << "region\_id:" << response.region\_meta().region\_id();

} else {

g\_error\_count << 1;

CHECK(brpc::IsAskedToQuit() || !FLAGS\_dont\_fail)

<< "error=" << cntl.ErrorText() << " latency=" << cntl.latency\_us();

cout << "error:" << cntl.ErrorText() << endl;

if (FLAGS\_sleep\_time >= 100000) {

bthread\_usleep(FLAGS\_sleep\_time);

} else {

bthread\_usleep(50000);

}

}

}

return (void \*)NULL;

}

在TDStore中使用如下：

void Commit(tdstore::TDStoreService\_Stub \*stub, const uint64\_t replication\_group\_id, const uint64\_t trans\_id,

const int request\_id, const int log\_id) {

brpc::Controller cntl;

tdstore::CommitRequest end\_trans\_request;

tdstore::CommitResponse end\_trans\_response;

end\_trans\_request.mutable\_head()->set\_request\_id(request\_id);

tdstore::ReplicationGroupRequestInfo \*participant = end\_trans\_request.add\_participants();

participant->set\_rep\_group\_id(replication\_group\_id);

end\_trans\_request.set\_trans\_id(trans\_id);

end\_trans\_request.mutable\_head()->set\_request\_id(request\_id);

cntl.set\_log\_id(log\_id); // set by user

stub->Commit(&cntl, &end\_trans\_request, &end\_trans\_response, NULL);

if (!cntl.Failed()) {

g\_latency\_recorder << cntl.latency\_us();

} else {

CHECK(brpc::IsAskedToQuit() || !FLAGS\_dont\_fail)

<< "error=" << cntl.ErrorText() << " latency=" << cntl.latency\_us();

if (FLAGS\_sleep\_time >= 100000) {

bthread\_usleep(FLAGS\_sleep\_time);

} else {

bthread\_usleep(100000);

}

}

}

在Record操作中PutRecord、GetRecord、ScanRecord使用：

void PutACRecord(tdstore::TDStoreService\_Stub \*stub, const uint64\_t region\_id, const uint64\_t trans\_id,

const int64\_t commit\_ts, const int request\_id, const int log\_id) {

brpc::Controller cntl;

tdstore::PutACRecordRequest put\_request;

tdstore::PutACRecordResponse put\_response;

std::string key = "key-" + std::to\_string(log\_id);

std::string value = "tdsql-" + std::to\_string(log\_id);

put\_request.mutable\_head()->set\_request\_id(trans\_id);

put\_request.set\_region\_id(1);

put\_request.set\_trans\_id(trans\_id);

put\_request.set\_key(key);

put\_request.set\_value(value);

put\_request.set\_schema\_obj\_id(g\_schema\_obj\_id);

put\_request.set\_schema\_obj\_version(g\_schema\_obj\_version);

cntl.set\_log\_id(log\_id); // set by user

cntl.set\_timeout\_ms(3000);

stub->PutACRecord(&cntl, &put\_request, &put\_response, NULL);

if (!cntl.Failed()) {

g\_latency\_recorder << cntl.latency\_us();

} else {

LOG(WARNING) << cntl.ErrorText();

CHECK(brpc::IsAskedToQuit() || !FLAGS\_dont\_fail)

<< "error=" << cntl.ErrorText() << " latency=" << cntl.latency\_us();

if (FLAGS\_sleep\_time >= 100000) {

bthread\_usleep(FLAGS\_sleep\_time);

} else {

bthread\_usleep(100000);

}

}

}

## Server端执行流程

bRPC Server流程梳理：<https://zhuanlan.zhihu.com/p/111490876>

<https://github.com/apache/brpc/blob/master/docs/cn/server.md>

Server Push：<https://github.com/apache/brpc/blob/master/docs/cn/server_push.md>

### 背景

### 原理

#### 处理一次RPC请求的完整过程

#### 服务器自动限流

#### 防雪崩

### 使用

## 并发读写TCP连接上的数据

### protobuf编程模式

### 多线程向同一TCP连接写入数据

#### 多线程向同一个TCP连接写数据的设计原理

考虑brpc自带的示例程序example/multi\_threaded\_echo\_c++/client.cpp，use\_bthread为true的情况下，多个bthread通过一条TCP长连接向服务端发送数据，而多个bthread通常又是运行在多个系统线程pthread上的，所以多个pthread如何高效且线程安全地向一个TCP连接写数据，是系统设计需要重点考虑的。brpc针对这个问题的设计思路如下：

1、为每个可被多线程写入数据的fd维护一个单项链表，每个试图向fd写数据的线程首先判断自己是不是当前第一个向fd写数据的线程，如果是，则持有写数据的权限，可以执行向fd写数据的操作；如果不是，则将待写数据加入链表就即刻返回（bthread执行结束，挂起，等待响应数据）。

2、掌握写权限的线程，在向fd写数据的时候，不仅可以写本线程持有的待写数据，而且可以观察到fd的链表上是否还加入了其他线程的待写数据，写入的时候可以尽量写入足够多的数据，但只执行一次写操作，如果因为fd的内核inode输出缓冲区已满而未能全部写完，则启动一个新的bthread去执行后续的写操作，当前bthread立即返回（被挂起，等待响应response的bthread唤醒）。

3、新启动的执行写操作的bthread，负责将fd的链表上的所有待写入数据写入fd（后续可能会有线程不断将待写数据加入待写链表），直到将链表清空。如果fd的内核inode缓冲区已满而不能写入，则该bthread将被挂起，让出cpu。等到epoll通知fd可写时，该thread再被唤起，继续写入。

4、KeepWrite bthread直到通过一个原子操作判断出\_write\_hread已为NULL时，才会执行完成，如果同时刻有一个线程通过原子操作判断出\_write\_hread为NULL，则重复上述过程1，所以不可能同时有两个KeepWrite bthread存在。

5、按照如上规则，所有bthread都不会有任何的等待操作，这就做到了wait-free，当然也是lock-free的（判断自己是不是第一个向fd写数据的线程的操作实际上是个原子交换操作）。

下面结合brpc的源码来阐述这套逻辑的实现。

#### brpc中的代码实现

brpc中的Socket类对象代表Client端与Server端的一条TCP连接，其中主要函数有：

* StartWrite函数：每个bthread向TCP连接写数据的入口，在实际环境下通常会被多个pthread执行，必须要做到线程安全：

int Socket::StartWrite(WriteRequest\* req, const WriteOptions& opt) {

// Release fence makes sure the thread getting request sees \*req

// 与当前\_write\_head做原子交换，\_write\_head初始值是NULL，

// 如果是第一个写fd的线程，则exchange返回NULL，并将\_write\_head指向第一个线程的待写数据，

// 如果不是第一个写fd的线程，exchange返回值是非NULL，且将\_write\_head指向最新到来的待写数据。

WriteRequest\* const prev\_head =

\_write\_head.exchange(req, butil::memory\_order\_release);

if (prev\_head != NULL) {

// Someone is writing to the fd. The KeepWrite thread may spin

// until req->next to be non-UNCONNECTED. This process is not

// lock-free, but the duration is so short(1~2 instructions,

// depending on compiler) that the spin rarely occurs in practice

// (I've not seen any spin in highly contended tests).

// 如果不是第一个写fd的bthread，将待写数据加入链表后，就返回。

req->next = prev\_head;

return 0;

}

int saved\_errno = 0;

bthread\_t th;

SocketUniquePtr ptr\_for\_keep\_write;

ssize\_t nw = 0;

// We've got the right to write.

// req指向的是第一个待写数据，肯定是以\_write\_head为头部的链表的尾结点，next一定是NULL。

req->next = NULL;

// Connect to remote\_side() if not.

// 如果TCP连接未建立，则在ConnectIfNot内部执行非阻塞的connect，并将自身挂起，

// 等待epoll通知连接已建立后再被唤醒执行。

int ret = ConnectIfNot(opt.abstime, req);

if (ret < 0) {

saved\_errno = errno;

SetFailed(errno, "Fail to connect %s directly: %m", description().c\_str());

goto FAIL\_TO\_WRITE;

} else if (ret == 1) {

// We are doing connection. Callback `KeepWriteIfConnected'

// will be called with `req' at any moment after

// TCP连接建立中，bthread返回、挂起，等待唤醒。

return 0;

}

// NOTE: Setup() MUST be called after Connect which may call app\_connect,

// which is assumed to run before any SocketMessage.AppendAndDestroySelf()

// in some protocols(namely RTMP).

req->Setup(this);

if (ssl\_state() != SSL\_OFF) {

// Writing into SSL may block the current bthread, always write

// in the background.

goto KEEPWRITE\_IN\_BACKGROUND;

}

// Write once in the calling thread. If the write is not complete,

// continue it in KeepWrite thread.

// 向fd写入数据，这里只关心req指向的数据，不关心其他bthread加入\_write\_head链表的数据。

// 不一定能一次写完，可能req指向的数据只写入了一部分。

if (\_conn) {

butil::IOBuf\* data\_arr[1] = { &req->data };

nw = \_conn->CutMessageIntoFileDescriptor(fd(), data\_arr, 1);

} else {

nw = req->data.cut\_into\_file\_descriptor(fd());

}

if (nw < 0) {

// RTMP may return EOVERCROWDED

if (errno != EAGAIN && errno != EOVERCROWDED) {

saved\_errno = errno;

// EPIPE is common in pooled connections + backup requests.

PLOG\_IF(WARNING, errno != EPIPE) << "Fail to write into " << \*this;

SetFailed(saved\_errno, "Fail to write into %s: %s",

description().c\_str(), berror(saved\_errno));

goto FAIL\_TO\_WRITE;

}

} else {

AddOutputBytes(nw);

}

// 判断req指向的数据是否已写完。

// 在IsWriteComplete内部会判断，如果req指向的数据已全部写完，且当前时刻req是唯一待写入的数据，

// 则IsWriteComplete返回true。

if (IsWriteComplete(req, true, NULL)) {

// 回收req指向的heap内存到对象池，bthread完成任务，返回。

ReturnSuccessfulWriteRequest(req);

return 0;

}

KEEPWRITE\_IN\_BACKGROUND:

ReAddress(&ptr\_for\_keep\_write);

req->socket = ptr\_for\_keep\_write.release();

// req指向的数据未全部写完，为了使pthread wait-free，启动KeepWrite bthread后，当前bthread就返回。

// 在KeepWrite bthread内部，不仅需要处理当前req未写完的数据，还可能要处理其他bthread加入链表的数据。

// KeepWrite bthread并不具有最高的优先级，所以使用bthread\_start\_background，将KeepWrite bthread的

// tid加到执行队列尾部。

if (bthread\_start\_background(&th, &BTHREAD\_ATTR\_NORMAL,

KeepWrite, req) != 0) {

LOG(FATAL) << "Fail to start KeepWrite";

KeepWrite(req);

}

return 0;

FAIL\_TO\_WRITE:

// `SetFailed' before `ReturnFailedWriteRequest' (which will calls

// `on\_reset' callback inside the id object) so that we immediately

// know this socket has failed inside the `on\_reset' callback

ReleaseAllFailedWriteRequests(req);

errno = saved\_errno;

return -1;

}

* KeepWrite函数：作为一个独立存在的bthread的任务函数，负责不停地写入所有线程加入到\_write\_hread链表的数据，直到链表为空：

void\* Socket::KeepWrite(void\* void\_arg) {

g\_vars->nkeepwrite << 1;

WriteRequest\* req = static\_cast<WriteRequest\*>(void\_arg);

SocketUniquePtr s(req->socket);

// When error occurs, spin until there's no more requests instead of

// returning directly otherwise \_write\_head is permantly non-NULL which

// makes later Write() abnormal.

WriteRequest\* cur\_tail = NULL;

do {

// req was written, skip it.

// 如果req的next指针不为NULL，则已经调用过IsWriteComplete实现了单向链表的翻转，

// 待写数据的顺序已按到达序排列。

// 所以如果req的next指针不为NULL且req的数据已写完，可以即刻回收req指向的内存，

// 并将req重新赋值为下一个待写数据的指针。

if (req->next != NULL && req->data.empty()) {

// 执行到这里，就是因为虽然req指向的WriteRequest中的数据已写完，

// 但\_write\_head链表中又被其他bthread加入了待写数据。

WriteRequest\* const saved\_req = req;

req = req->next;

s->ReturnSuccessfulWriteRequest(saved\_req);

}

// 向fd写入一次数据，DoWrite内部的实现为尽可能的多谢，可以连带req后面的待写数据一起写。

const ssize\_t nw = s->DoWrite(req);

if (nw < 0) {

if (errno != EAGAIN && errno != EOVERCROWDED) {

// 如果不是因为内核inode输出缓存已满导致的write操作结果小于0，

// 则标记Socket对象状态异常（TCP连接异常）。

const int saved\_errno = errno;

PLOG(WARNING) << "Fail to keep-write into " << \*s;

s->SetFailed(saved\_errno, "Fail to keep-write into %s: %s",

s->description().c\_str(), berror(saved\_errno));

break;

}

} else {

s->AddOutputBytes(nw);

}

// Release WriteRequest until non-empty data or last request.

// 可能一次写入了链表中多个节点中的待写数据，数据已写完的节点回收内存。

// while操作结束后req指向的是已翻转的链表中的第一个数据未写完的节点。

while (req->next != NULL && req->data.empty()) {

WriteRequest\* const saved\_req = req;

req = req->next;

s->ReturnSuccessfulWriteRequest(saved\_req);

}

// TODO(gejun): wait for epollout when we actually have written

// all the data. This weird heuristic reduces 30us delay...

// Update(12/22/2015): seem not working. better switch to correct code.

// Update(1/8/2016, r31823): Still working.

// Update(8/15/2017): Not working, performance downgraded.

//if (nw <= 0 || req->data.empty()/\*note\*/) {

if (nw <= 0) {

// 执行到这里，nw小于0的原因肯定是因为内核inode输出缓存已满。

// 如果是由于fd的inode输出缓冲区已满导致write操作返回值小于等于0，则需要挂起执行KeepWrite的

// bthread，让出cpu，让该bthread所在的pthread去任务队列中取出下一个bthread去执行。

// 等到epoll返回告知inode输出缓冲区有可写空间时，再唤起执行KeepWrite的bthread，继续向fd写入数据。

g\_vars->nwaitepollout << 1;

bool pollin = (s->\_on\_edge\_triggered\_events != NULL);

// NOTE: Waiting epollout within timeout is a must to force

// KeepWrite to check and setup pending WriteRequests periodically,

// which may turn on \_overcrowded to stop pending requests from

// growing infinitely.

const timespec duetime =

butil::milliseconds\_from\_now(WAIT\_EPOLLOUT\_TIMEOUT\_MS);

// 在WaitEpollOut内部会执行butex\_wait，挂起当前bthread。当bthread重新执行时，执行点是

// butex\_wait的函数返回点。

const int rc = s->WaitEpollOut(s->fd(), pollin, &duetime);

if (rc < 0 && errno != ETIMEDOUT) {

const int saved\_errno = errno;

PLOG(WARNING) << "Fail to wait epollout of " << \*s;

s->SetFailed(saved\_errno, "Fail to wait epollout of %s: %s",

s->description().c\_str(), berror(saved\_errno));

break;

}

}

// 令cur\_tail找到已翻转链表的尾节点。

if (NULL == cur\_tail) {

for (cur\_tail = req; cur\_tail->next != NULL;

cur\_tail = cur\_tail->next);

}

// 执行到这里，cur\_tail指向的是当前已被翻转的链表的尾节点。

// Return when there's no more WriteRequests and req is completely

// written.

if (s->IsWriteComplete(cur\_tail, (req == cur\_tail), &cur\_tail)) {

// 如果IsWriteComplete返回true，则req必然，并且当前的\_write\_hread肯定是NULL。

CHECK\_EQ(cur\_tail, req);

// 回收内存后KeepWrite bthread就结束了，后续再有线程向fd写数据，则重复以前的逻辑。

// 所以同一时刻只会存在一个KeepWrite bthread。

s->ReturnSuccessfulWriteRequest(req);

return NULL;

}

} while (1);

// Error occurred, release all requests until no new requests.

s->ReleaseAllFailedWriteRequests(req);

return NULL;

}

* IsWriteComplete函数：两种情况下会调用IsWriteComplete函数，1、持有写权限的bthread向fd写自身的WriteRequest中的待写数据，写一次fd后检测自身的WriteRequest中的数据是否写完；2、被KeepWrite bthread中执行，检测上一轮经过翻转的单向链表中的各个WriteRequest中数据是否全部写完。并且IsWriteComplete函数内部还负责检测是否还有其他bthread向\_write\_head链表加入了新的待写数据。

bool Socket::IsWriteComplete(Socket::WriteRequest\* old\_head,

bool singular\_node,

Socket::WriteRequest\*\* new\_tail) {

// old\_head只有两种可能：1、指向持有写权限的bthread携带的WriteRequest，2、指向上一轮经过翻转的链表的尾节点。

// 不论是哪两种，old\_head指向的WriteRequest的next必然是NULL。

CHECK(NULL == old\_head->next);

// Try to set \_write\_head to NULL to mark that the write is done.

WriteRequest\* new\_head = old\_head;

WriteRequest\* desired = NULL;

bool return\_when\_no\_more = true;

if (!old\_head->data.empty() || !singular\_node) {

desired = old\_head;

// Write is obviously not complete if old\_head is not fully written.

return\_when\_no\_more = false;

}

// 1、如果之前翻转的链表已全部写完，则将\_write\_head置为NULL，当前的KeepWrite bthread也即将结束；

// 2、如果之前翻转的链表未全部写完，且暂时无其他bthread向\_write\_head新增待写数据，\_write\_head指针保存原值；

// 3、如果之前翻转的链表未全部写完，且已经有其他bthread向\_write\_head新增待写数据，将new\_head的值置为当前

// 最新的\_write\_head值，为后续的链表翻转做准备。

if (\_write\_head.compare\_exchange\_strong(

new\_head, desired, butil::memory\_order\_acquire)) {

// No one added new requests.

if (new\_tail) {

\*new\_tail = old\_head;

}

return return\_when\_no\_more;

}

// 执行到这里，一定是有其他bthread将待写数据加入到了\_write\_head链表里，

// 经过compare\_exchange\_strong后new\_head指向当前\_write\_head所指的WriteRequest实例，肯定是不等于old\_head的。

CHECK\_NE(new\_head, old\_head);

// Above acquire fence pairs release fence of exchange in Write() to make

// sure that we see all fields of requests set.

// Someone added new requests.

// Reverse the list until old\_head.

// 将以new\_head为头节点、old\_head为尾节点的单向链表做一次翻转，保证待写数据以先后顺序排序。

// 随时可能有新的bthread将待写数据加入到\_write\_head链表，但暂时不考虑这些新来的数据。

WriteRequest\* tail = NULL;

WriteRequest\* p = new\_head;

do {

while (p->next == WriteRequest::UNCONNECTED) {

// TODO(gejun): elaborate this

sched\_yield();

}

WriteRequest\* const saved\_next = p->next;

p->next = tail;

tail = p;

p = saved\_next;

CHECK(p != NULL);

} while (p != old\_head);

// Link old list with new list.

old\_head->next = tail;

// Call Setup() from oldest to newest, notice that the calling sequence

// matters for protocols using pipelined\_count, this is why we don't

// calling Setup in above loop which is from newest to oldest.

for (WriteRequest\* q = tail; q; q = q->next) {

q->Setup(this);

}

// 将\*new\_tail指向当前最新的已翻转链表的尾节点。

if (new\_tail) {

\*new\_tail = new\_head;

}

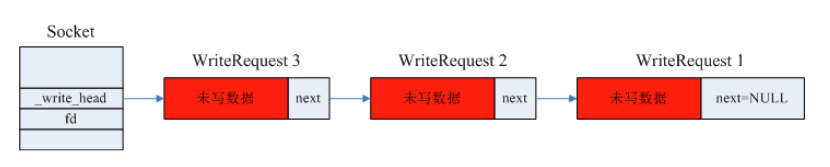
return false;

}

#### 一个实际场景下的示例

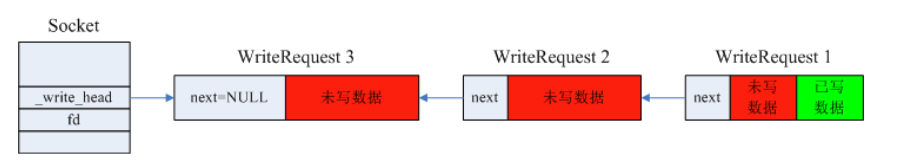
下面以一个实际场景为例，说明线程执行过程和内存变化过程：

1、假设T0时刻有3个分别被不同pthread执行的bthread同时向同一个fd写入数据，3个bthread同时进入到StartWrite函数执行\_write\_head.exchange原子操作，\_write\_head初始值是NULL，假设bthread 0第一个用自己的req指针与\_write\_head做exchange，则bthread 0获取了向fd写数据的权限，bthread 1和bthread 2将待发送的数据加入\_write\_head链表后直接return 0返回（bthread 1和bthread 2返回后会被挂起，yield让出cpu）。此时内存结构为：



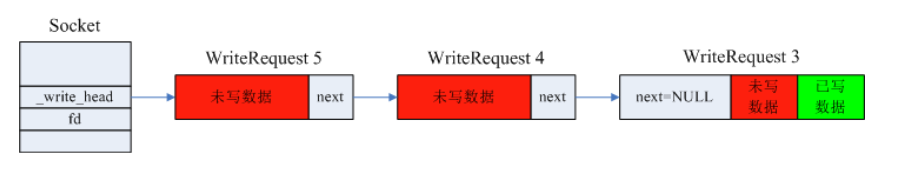
2、T1时刻起（后续若无特别说明，假设暂时没有新的bthread再往\_write\_head链表中加入待写数据），bthread 1向fd写自身携带的WriteRequest 1中的数据（假设TCP长连接已建好，在ConnectIfNot内部不发起非阻塞的connect调用），执行一次写操作后，进入IsWriteComplete，判断是否写完（WriteRequest 1中的数据未写完，或者虽然WriteRequest 1的数据写完了但是还有其他bthread往链表中加入了待写数据，都算没写完。本示例中此时IsWriteComplete肯定是返回false的）。

3、bthread 1所在的pthread执行进入IsWriteComplete（假设WriteRequest 1中的数据没有全部写完），在IsWriteComplete中判断出WriteRequest 1中仍然有未写数据，并且\_write\_head也并不指向WriteRequest 1而是指向了新来的WriteRequest 3，为保证将数据依先后顺序写入fd，将图1所示的单向链表做翻转（代码中的\_write\_head.compare\_exchange\_strong操作的是最新的\_write\_head，在这个原子操作后，仍然会有bthread将待写数据加入到\_write\_head，\_write\_head会变化。但上述的链表翻转之后，如果有新来的WriteRequest暂时也不管它，后续会处理。这里先假设没有bthread加入新的待写数据）。此时内存结构为：

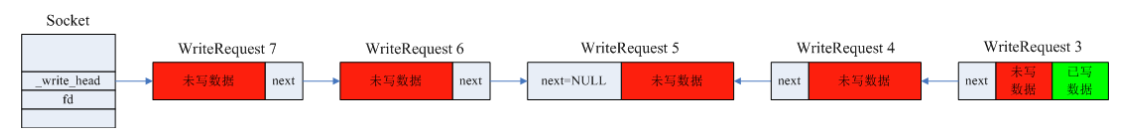


4、因为IsWriteComplete返回了false，仍然有待写数据要写，但接下来的写操作不能再由bthread 1负责，因为剩下的待写数据也不能保证一次都写完，bthread 1不可能去等待fd的内核inode输出缓存是否有可用空间，否则会令bthread 1所在的整个pthread线程卡顿，pthread私有的TaskGroup上的任务执行队列中其他bthread就得不到及时执行了，也就不是wait-free了。因此bthread 1创建一个新的KeepWrite bthread专门负责剩余数据的发送，bthread 1即刻返回（bthread 1到这里也就完成了任务，会被挂起，yield让出cpu）。

5、T3时刻起，KeepWrite bthread得到了调度，被某一个pthread执行，开始写之前剩余的数据。假设一次向fd的写操作执行后，WriteRequest 1、WriteRequest 2中的数据全部写完（WriteRequest对象随即被回收内存），WriteRequest 3写了一部分，并且此时又有其他两个bthread向\_write\_head链表中新加入了待写数据WriteRequest 4和WriteRequest 5，此时内存结构为：



6、KeepWrite bthread执行IsWriteComplete判断之前已翻转过的链表是否已全部写完，并在IsWriteComplete内部通过\_write\_head.compare\_exchange\_strong原子操作检测到之前新增的待写数据WriteRequest 4、5，并完成WriteRequest 5->4->3链表的翻转。假设在\_write\_head.compare\_exchange\_strong执行之后立即有其他bthread又向\_write\_head链表中新加入了待写数据WriteRequest 6和WriteRequest 7，但WriteRequest 6和7在\_write\_head.compare\_exchange\_strong一旦被调用之后是暂时被无视的，等到下一轮调用IsWriteComplete时才会被发现（通过\_write\_head.compare\_exchange\_strong发现最新的\_write\_head不等于之前已被翻转的链表的尾节点）。此时内存结构为：



7、KeepWrite bthread调用IsWriteComplete返回后，cur\_tail指针指向的是当前最新的已翻转链表的尾节点，即WriteRequest 5。然后继续尽可能多的向fd写一次数据，一次写操作后再调用IsWriteComplete判断当前已翻转的链表是否已全部写完、\_write\_head链表是否有新加入的待写数据，会发现之前新加入的WriteRequest 6和WriteRequest 7，再将WriteRequest 6、7加入翻转链表...如此循环反复。只有在IsWriteComplete中判断出已翻转链表的待写数据已全部写入fd、且当前最新的\_write\_head指向翻转链表的尾节点时，KeepWrite bthread才会执行结束。如果紧接着又有其他bthread试图向fd写入数据，则重复步骤1开始的过程。所以同一时刻针对一个TCP连接不会同时存在多个KeepWrite bthread，一个TCP连接上最多只会有一个KeepWrite bthread在工作。

8、KeepWrite bthread在向fd写数据时，fd一般被设为非阻塞，如果fd的内核inode输出缓存已满，对fd调用::write（或者::writev）会返回-1且errno为EAGAIN。这时候KeepWrite bthread不能去主动等待fd是否可写，必须要挂起，yield让出cpu，让KeepWrite bthread所在的pthread接着去调度执行私有的TaskGroup的任务队列中的下一个bthread，否则pthread就会卡住，影响了TaskGroup任务队列中其他bthread的执行，这违背了wait-free的设计思想。等到内核inode输出缓存有了可用空间时，epoll会返回fd的可写事件，epoll所在的bthread会唤醒KeepWrite bthread，KeepWrite bthread的id会被重新加入某个TaskGroup的任务队列，从而会被重新调度执行，继续向fd写入数据。

### 从TCP连接读取数据的并发处理

#### 传统RPC框架从fd读取数据的方式

传统的RPC框架一般会区分I/O线程和worker线程，一台机器上处理的所有fd会散列到多个I/O线程上，每个I/O线程在其私有的EventLoop对象上执行类似下面这样的循环：

while (!stop) {

int n = epoll\_wait();

for (int i = 0; i < n; ++i) {

if (event is EPOLLIN) {

// read data from fd

// push pointer of reference of data to task queue

}

}

}

I/O线程从每个fd上读取到数据后，将已读取数据的指针或引用封装在一个Task对象中，再将Task的指针压入一个全局的任务队列，worker线程从任务队列中拿到报文并进行业务处理。

这种方式存在以下几个问题：

1、一个I/O线程同一时刻只能从一个fd读取数据，数据从fd的inode内核缓存读取到应用层缓冲区是一个相对较慢的操作，读取顺序越靠后的fd，其上面的数据读取、处理越可能产生延迟。实际场景中，如果10个客户端同一时刻分别通过10条TCP连接向一个服务器发送请求，假设服务器只开启一个I/O线程，epoll同时通知10个fd有数据可读，I/O线程只能先读完第一个fd，再读去读第二个fd...可能等到读第9、第10个fd时，客户端已经报超时了。但实际上10个客户端是同一时刻发送的请求，服务器的读取数据顺序却有先有后，这对客户端来说是不公平的；

2、多个I/O线程都会将Task压入一个全局的任务队列，会产生锁竞争；多个worker线程从全局任务队列中获取任务，也会产生锁竞争，这都会降低性能。并且多个worker线程从长久看很难获取到均匀的任务数量，例如有4个worker线程同时去任务队列拿任务，worker 1竞争锁成功，拿到任务后去处理，然后worker 2拿到任务，接着worker 3拿到任务，再然后worker 1可能已经处理任务完毕，又来和worker 4竞争全局队列的锁，可能worker 1又再次竞争成功，worker 4还是饿着，这就造成了任务没有在各个worker线程间均匀分配；

3、I/O线程从fd的inode内核缓存读数据到应用层缓冲区，worker线程需要处理这块内存上的数据，内存数据同步到执行worker线程的cpu的cacheline上需要耗费时间。

#### brpc实现的读数据方式

brpc没有专门的I/O线程，只有worker线程，epoll\_wait()也是在bthread中被执行。当一个fd可读时：

1、读取动作并不是在epoll\_wait()所在的bthread 1上执行，而是会通过TaskGroup::start\_foreground()新建一个bthread 2，bthread 2负责将fd的inode内核输入缓存中的数据读到应用层缓存区，pthread执行流会立即进入bthread 2，bthread 1会被加入任务队列的尾部，可能会被steal到其他pthread上执行；

2、bthread 2进行拆包时，每解析出一个完整的应用层报文，就会为每个报文的处理再专门创建一个bthread，所以bthread 2可能会创建bthread 3、4、5...这样的设计意图是尽量让一个fd上读出的各个报文也得到最大化的并发处理。

## 内存管理

参考：

<https://github.com/apache/brpc/blob/master/docs/cn/memory_management.md>

### ResourcePool：多线程下高效的内存分配与回收

#### 多线程下内存分配与回收的设计原则

多线程高度竞争环境下的内存分配，需要考虑两个主要问题：

如何尽可能地避免内存分配过程和回收过程的线程间的竞争。brpc的ResourcePool的设计方案是：

1、为每个线程维护一个私有的内存区和一个空闲对象列表；

当一个线程需要为某个新建对象分配内存时，先假设此时线程私有的、以及全局的空闲对象列表都为空，线程会从自己的私有内存区上分配，私有内存区用满后，从全局内存区再次申请一块内存作为新的私有内存区。所以为新建对象分配内存的动作不存在线程间的竞争，不需要用锁。只有从全局内存区申请一块线程私有内存时可能存在竞态，需要加锁；

当一个对象不再被使用时，不析构该对象，不释放对象占用的内存，只是将对象的唯一id（不仅是对象的唯一标识id，而且通过此id能在O(1)时间内定位到对象的内存地址，见下文描述）存入线程的私有空闲对象列表。当私有空闲对象列表已满后，将已满的空闲对象列表拷贝一份，将拷贝列表压入全局的空闲列表队列，并清空线程私有的空闲对象列表。所以回收对象内存的动作不存在线程间的竞争，不需要用锁，只有将已满的空闲对象列表压入全局的空闲列表队列时需要加锁。线程为新建对象分配内存时，首先查看私有的空闲对象列表，如果不为空，则取出一个对象id，通过id找到一个之前已回收的对象，重置对象状态，作为新对象使用；如果私有空闲对象列表为空，则再尝试从全局的空闲列表队列中弹出一个空闲列表，将列表内容拷贝到私有空闲对象列表，再从私有空闲对象列表拿出一个已回收对象的id；如果全局的空闲列表队列也为空，则再从线程私有内存区上分配一块内存；

brpc的ResourcePool的分配、回收对象内存的方式不解决ABA问题，例如，线程A使用对象obj后将该对象回收，一段时间后线程B拿到了obj，重置obj的状态后作为自己的对象使用，此时线程A用obj的id仍然能访问到obj，但obj的所有权已不属于线程A了，线程A若再操作obj会导致程序混乱。所以需要在应用程序中使用版本号等方法防止内存分配回收的ABA问题。

2、如何避免内存碎片。brpc的ResourcePool为每一个类型的对象（TaskMeta、Socket、Id等）单独建立一个全局的singleton内存区，每个singleton内存区上分配的对象都是等长的，所以分配、回收过程不会有内存碎片。

#### ResourcePool的内存布局

一个ResourcePool单例对象表示某一个类型的对象的全局singleton内存区。先解释下ResourcePool中的成员变量和几个宏所表达的意义：

1、RP\_MAX\_BLOCK\_NGROUP表示一个ResourcePool中BlockGroup的数量；RP\_GROUP\_NBLOCK表示一个BlockGroup中的Block\*的数量；

2、ResourcePool是模板类，模板参数是对象类型，主要的成员变量：

1）\_local\_pool：一个ResourcePool单例上，每个线程有各自私有的LocalPool对象，\_local\_pool是私有LocalPool对象的指针。LocalPool类的成员有：

\_cur\_block：从ResourcePool单例的全局内存区申请到的一块Block的指针。

\_cur\_block\_index：\_cur\_block指向的Block在ResourcePool单例的全局内存区中的Block索引号。

\_cur\_free：LocalPool对象中未存满被回收对象id的空闲对象列表。

2）\_nlocal：一个ResourcePool单例上，LocalPool对象的数量。

3）\_block\_groups：一个ResourcePool单例上，BlockGroup指针的数组。一个BlockGroup中含有RP\_GROUP\_NBLOCK个Block的指针。一个Block中含有若干个对象。

4）\_ngroup：一个ResourcePool单例上，BlockGroup的数量。

5）\_free\_chunks：一个ResourcePool单例上，已经存满被回收对象id的空闲列表的队列。

一个ResourcePool对象的内存布局如下图表示：

TODO

#### ResourcePool的源码解析

为对象分配内存和回收内存的主要代码都在ResourcePool类中。

1、针对一种类型的对象，获取全局的ResourcePool的单例的接口函数为ResourcePool::singleton()，该函数可被多个线程同时执行，要注意代码中的double check逻辑判断：

static inline ResourcePool\* singleton() {

// 如果当前\_singleton指针不为空，则之前已经有线程为\_singleton赋过值，直接返回非空的\_singleton值即可。

ResourcePool\* p = \_singleton.load(butil::memory\_order\_consume);

if (p) {

return p;

}

// 修改\_singleton的代码可被多个线程同时执行，必须先加锁。

pthread\_mutex\_lock(&\_singleton\_mutex);

// double check，再次检查\_singleton指针是否为空。

// 因为可能有两个线程同时进入ResourcePool::singleton()函数，同时检测到\_singleton值为空，

// 接着同时执行到pthread\_mutex\_lock(&\_singleton\_mutex)，但只能有一个线程（A）执行\_singleton.store()，

// 另一个线程（B）必须等待。线程A执行pthread\_mutex\_unlock(&\_singleton\_mutex)后，线程B恢复执行，必须再次

// 判断\_singleton是否为空，因为\_singleton之前已经被线程A赋了值，线程B不能再次给\_singleton赋值。

p = \_singleton.load(butil::memory\_order\_consume);

if (!p) {

// 创建一个新的ResourcePool对象，对象指针赋给\_singleton。

p = new ResourcePool();

\_singleton.store(p, butil::memory\_order\_release);

}

pthread\_mutex\_unlock(&\_singleton\_mutex);

return p;

}

2、为一个指定类型的对象分配内存的接口函数为ResourcePool::get\_resource()，根据对象的不同构造函数，get\_resource()有相应的几个重载，逻辑都为先调用get\_or\_new\_local\_pool()取得线程私有的LocalPool对象，再调用LocalPool对象的get()函数，get()函数的内容封装在BAIDU\_RESOURCE\_POOL\_GET宏中：

#define BAIDU\_RESOURCE\_POOL\_GET(CTOR\_ARGS) \

/\* Fetch local free id \*/ \

// 如果线程私有的空闲对象列表当前存有空闲对象，则直接从空闲列表拿出一个空闲对象的id。

if (\_cur\_free.nfree) { \

const ResourceId<T> free\_id = \_cur\_free.ids[--\_cur\_free.nfree]; \

\*id = free\_id; \

BAIDU\_RESOURCE\_POOL\_FREE\_ITEM\_NUM\_SUB1; \

// 使用对象id在O(1)时间内定位到空闲对象的地址，并返回。

return unsafe\_address\_resource(free\_id); \

} \

/\* Fetch a FreeChunk from global. \

TODO: Popping from \_free needs to copy a FreeChunk which is \

costly, but hardly impacts amortized performance. \*/ \

// 线程私有的空闲对象列表没有空闲对象可用，从全局的空闲列表队列中弹出一个空闲对象列表，

// 弹出的空闲对象列表存满了已被回收的空闲对象的id，将弹出的空闲对象列表的内容拷贝到线程私有空闲对象列表。

if (\_pool->pop\_free\_chunk(\_cur\_free)) { \

// 此时线程私有的空闲对象列表存满了空闲对象的id，现在可以从中拿出一个空闲对象的id。

--\_cur\_free.nfree; \

const ResourceId<T> free\_id = \_cur\_free.ids[\_cur\_free.nfree]; \

\*id = free\_id; \

BAIDU\_RESOURCE\_POOL\_FREE\_ITEM\_NUM\_SUB1; \

// 使用对象id在O(1)时间内定位到空闲对象的地址，并返回。

return unsafe\_address\_resource(free\_id); \

} \

/\* Fetch memory from local block \*/ \

// 全局都没有空闲对象可用，如果此时线程私有的Block中有剩余空间可用，则从私有的Block中分配一块内存给新建对象。

if (\_cur\_block && \_cur\_block->nitem < BLOCK\_NITEM) { \

// \_cur\_block\_index是当前的Block在全局所有的Block中的索引号，

// BLOCK\_NITEM是一个Block中最大能存储的对象的个数（一个Block中存储的所有对象肯定是同一类型的），

// \_cur\_block->nitem是线程私有的Block中已经分配内存的对象个数（对象可能是正在使用，也可能是已经被回收），

// 所以一个新分配内存的对象的id值就等于当前全局已被分配内存的所有对象的个数，即新对象所占内存空间在全局

// 对象池中的偏移量。

id->value = \_cur\_block\_index \* BLOCK\_NITEM + \_cur\_block->nitem; \

T\* p = new ((T\*)\_cur\_block->items + \_cur\_block->nitem) T CTOR\_ARGS; \

if (!ResourcePoolValidator<T>::validate(p)) { \

p->~T(); \

return NULL; \

} \

++\_cur\_block->nitem; \

return p; \

} \

/\* Fetch a Block from global \*/ \

// 线程私有的Block已用满，再从该类型对象的全局内存区中再申请一块Block。

\_cur\_block = add\_block(&\_cur\_block\_index); \

if (\_cur\_block != NULL) { \

id->value = \_cur\_block\_index \* BLOCK\_NITEM + \_cur\_block->nitem; \

T\* p = new ((T\*)\_cur\_block->items + \_cur\_block->nitem) T CTOR\_ARGS; \

if (!ResourcePoolValidator<T>::validate(p)) { \

p->~T(); \

return NULL; \

} \

++\_cur\_block->nitem; \

return p; \

} \

return NULL; \

ResourcePool::LocalPool::unsafe\_address\_resource()函数的作用为通过一个对象id在O(1)时间内定位到该对象的地址：

static inline T\* unsafe\_address\_resource(ResourceId<T> id) {

const size\_t block\_index = id.value / BLOCK\_NITEM;

return (T\*)(\_block\_groups[(block\_index >> RP\_GROUP\_NBLOCK\_NBIT)]

.load(butil::memory\_order\_consume)

->blocks[(block\_index & (RP\_GROUP\_NBLOCK - 1))]

.load(butil::memory\_order\_consume)->items) +

id.value - block\_index \* BLOCK\_NITEM;

}

ResourcePool::add\_block\_group()函数作用是新建一个BlockGroup并加入ResourcePool单例的\_block\_groups数组：

static bool add\_block\_group(size\_t old\_ngroup) {

BlockGroup\* bg = NULL;

BAIDU\_SCOPED\_LOCK(\_block\_group\_mutex);

const size\_t ngroup = \_ngroup.load(butil::memory\_order\_acquire);

if (ngroup != old\_ngroup) {

// Other thread got lock and added group before this thread.

return true;

}

if (ngroup < RP\_MAX\_BLOCK\_NGROUP) {

bg = new(std::nothrow) BlockGroup;

if (NULL != bg) {

// Release fence is paired with consume fence in address() and

// add\_block() to avoid un-constructed bg to be seen by other

// threads.

\_block\_groups[ngroup].store(bg, butil::memory\_order\_release);

\_ngroup.store(ngroup + 1, butil::memory\_order\_release);

}

}

return bg != NULL;

}

ResourcePool::add\_block()函数作用是新建一个Block并将Block的地址加入当前未满的BlockGroup的blocks数组：

static Block\* add\_block(size\_t\* index) {

Block\* const new\_block = new(std::nothrow) Block;

if (NULL == new\_block) {

return NULL;

}

size\_t ngroup;

do {

ngroup = \_ngroup.load(butil::memory\_order\_acquire);

if (ngroup >= 1) {

BlockGroup\* const g =

\_block\_groups[ngroup - 1].load(butil::memory\_order\_consume);

const size\_t block\_index =

g->nblock.fetch\_add(1, butil::memory\_order\_relaxed);

if (block\_index < RP\_GROUP\_NBLOCK) {

g->blocks[block\_index].store(

new\_block, butil::memory\_order\_release);

\*index = (ngroup - 1) \* RP\_GROUP\_NBLOCK + block\_index;

return new\_block;

}

g->nblock.fetch\_sub(1, butil::memory\_order\_relaxed);

}

} while (add\_block\_group(ngroup));

// Fail to add\_block\_group.

delete new\_block;

return NULL;

}

ResourcePool::pop\_free\_chunk()函数作用是从全局的存满空闲对象id的空闲列表队列中弹出一个空闲对象列表，将其内容拷贝到线程私有的空闲对象列表中（拷贝前线程私有的空闲对象列表一定是空的）：

bool pop\_free\_chunk(FreeChunk& c) {

// Critical for the case that most return\_object are called in

// different threads of get\_object.

if (\_free\_chunks.empty()) {

return false;

}

pthread\_mutex\_lock(&\_free\_chunks\_mutex);

if (\_free\_chunks.empty()) {

pthread\_mutex\_unlock(&\_free\_chunks\_mutex);

return false;

}

DynamicFreeChunk\* p = \_free\_chunks.back();

\_free\_chunks.pop\_back();

pthread\_mutex\_unlock(&\_free\_chunks\_mutex);

c.nfree = p->nfree;

memcpy(c.ids, p->ids, sizeof(\*p->ids) \* p->nfree);

free(p);

return true;

}

3、回收对象内存的接口函数为ResourcePool::return\_resource()，内部会调用LocalPool::return\_resource()函数：

inline int return\_resource(ResourceId<T> id) {

// Return to local free list

if (\_cur\_free.nfree < ResourcePool::free\_chunk\_nitem()) {

\_cur\_free.ids[\_cur\_free.nfree++] = id;

BAIDU\_RESOURCE\_POOL\_FREE\_ITEM\_NUM\_ADD1;

return 0;

}

// Local free list is full, return it to global.

// For copying issue, check comment in upper get()

// ResourcePool::push\_free\_chunk()函数的作用是

if (\_pool->push\_free\_chunk(\_cur\_free)) {

\_cur\_free.nfree = 1;

\_cur\_free.ids[0] = id;

BAIDU\_RESOURCE\_POOL\_FREE\_ITEM\_NUM\_ADD1;

return 0;

}

return -1;

}

### I/O读写缓冲区

## 实时监控

### bvar库

### 常用性能监控指标

## 基础库

### 侵入式双向链表

#### 侵入式链表原理及实现

侵入式链表与一般链表不同之处在于，将负责链表节点链接关系的next、prev指针单独拿出来实现为一个基类，在基类中完成链表节点的增、删等操作，具体业务相关的数据结构只需要继承这个基类，即可实现业务相关的链表，不需要在业务数据结构中实现增、删等操作。

brpc的侵入式链表相关代码在src/butil/containers/linked\_list.h中，next、prev指针和链表节点的增、删操作都定义在LinkNode模板类中，代码也很简单：

template <typename T>

class LinkNode {

public:

// LinkNode are self-referential as default.

LinkNode() : previous\_(this), next\_(this) {}

LinkNode(LinkNode<T>\* previous, LinkNode<T>\* next)

: previous\_(previous), next\_(next) {}

// Insert |this| into the linked list, before |e|.

void InsertBefore(LinkNode<T>\* e) {

// ......

}

// Insert |this| as a circular linked list into the linked list, before |e|.

void InsertBeforeAsList(LinkNode<T>\* e) {

// ......

}

// Insert |this| into the linked list, after |e|.

void InsertAfter(LinkNode<T>\* e) {

// ......

}

// Insert |this| as a circular list into the linked list, after |e|.

void InsertAfterAsList(LinkNode<T>\* e) {

// ......

}

// Remove |this| from the linked list.

void RemoveFromList() {

// ......

}

LinkNode<T>\* previous() const {

return previous\_;

}

LinkNode<T>\* next() const {

return next\_;

}

private:

LinkNode<T>\* previous\_;

LinkNode<T>\* next\_;

};

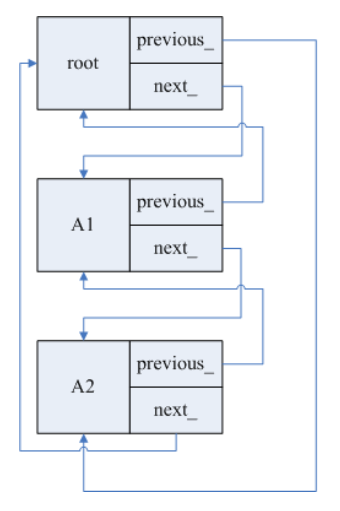
在具体业务中，想要实现一个数据类型为A的侵入式链表，只需要直接继承LinkNode基类就可以了：

class A : public LinkNode<A> {

// 定义A的成员变量

}

A的侵入式链表的内存如下图所示：



#### 侵入式链表在brpc中的应用

Butex锁的等待队列实质上是一个侵入式链表，只需注意每一个ButexWaiter对象都分配在各自bthread的私有栈上，而Butex对象本身分配在heap内存区的ResourcePool中。

### FlatMap哈希表

### 多线程框架下的定时器

# 应用

## OSS协程改造方案

### 背景

#### oss线程的使用

1、业务方面

当前oss使用的是线程池来处理每一条业务连接请求。线程池中线程数的设置通过scheduler.xml的thread属性来设置。

2、其余线程

oss日志清理线程

oss日志入监控库线程

配置热更新线程等.......

#### 当前oss构架

libevent +线程池+状态机

线程池中的每一条线程都有自己的libevent\_base

每一条http连接通过RR方式分配到线程池中的一条线程，注册libevent事件绑定回调函数

回调函数通过状态机逻辑走业务处理流程

也就是说一条http请求连接是对应OSS线程池中的一条线程来进行处理。

#### zk的访问

oss整个程序会初始化两个zk客户端（公共库的zk客户端为基础进行封装）:

用于业务处理的zk客户端

用于配置热更新的zk客户端

这里因为每条业务流程都要使用唯一的zk客户端，是加锁请求的。

#### 瓶颈

##### 接口超时

4个get接口（GetInstance,QueryGWInstance,GetGroup,GetGWGroup）等会有大量访问zk的逻辑，因此上述接口耗时很长。当线程池中所有的线程都在处理这种访问zk多耗时长的请求时，后续请求将在队列中等待，进而很容易超时。

由于只有一个zk连接，每条线程处理逻辑只要存在zk访问都要请求zk客户端的锁，并等待。尤其是上述get接口。

##### 扁鹊接口hang住线程

clouddba接口：

oss连接扁鹊，tcp连接成功后，扁鹊（故障）不返回数据，导致oss该线程在recv处hang住。

### 改造方案

#### libgo协程库介绍

libgo有以下特点：

1、提供golang一般功能强大协程，基于corontine编写代码，可以以同步的方式编写简单的代码，同时获得异步的性能

2、支持海量协程, 创建100万个协程只需使用2GB物理内存

3、允许用户自由控制协程调度点，随时随地变更调度线程数

4、支持多线程调度协程，极易编写并行代码，高效的并行调度算法，可以有效利用多个CPU核心

5、可以让链接进程序的同步的第三方库变为异步调用，大大提升其性能。再也不用担心某些DB官方不提供异步driver了，比如hiredis、mysqlclient这种客户端驱动可以直接使用，并且可以得到不输于异步driver的性能

6、动态链接和静态链接全都支持，便于使用C++11的用户静态链接生成可执行文件并部署至低版本的linux系统上

7、提供协程锁(co\_mutex), 定时器, channel等特性, 帮助用户更加容易地编写程序.

8、网络性能强劲，超越boost.asio异步模型；尤其在处理小包和多线程并行方面非常强大

##### oss使用libgo协程库期望的效果

* 将每条http请求将对应一条协程
* hook掉zk的调用（get/set），同步的zk调用变成异步

当该协程的zk（get/set）请求等待时，会发生协程切换

* 与扁鹊的连接，recv也将被hook掉，即扁鹊故障，与扁鹊通信的协程不会hang住（会发生协程的切换）

综上：当耗时长的接口在执行时，由于会访问zk并等待，此时会发生协程切换，将线程让出去处理其他的就绪协程，减小接口超时概率。

#### 公共库修改

##### 公共库添加libgo库

公共库添加开源的libgo 库并且将关键字go进行重定义define。

编译出库文件，提供给oss编译进行连接

##### 公共库zk客户端改造

* 公共库zookeeper\_acl中的线程锁和线程条件变量替换（使用#ifdefine）为协程锁co\_mutex和协程条件变量co\_condition\_variable
* 公共库zookeeper\_acl的编译通过传参-D进行不同版本编译

产生使用协程锁的zk库提供oss使用

产生原本的线程锁的zk库提供其他模块使用

#### OSS改造方案

##### 线程池与libevent回调构架

原先的线程池取消，每一条连接请求改为一个协程处理。

所有的协程交由一个libgo的scheduler调度器进行调度分配给任务线程

任务线程的数量通过scheduler.xml进行配置

##### 线程换成协程

目前使用线程的地方：

监控库入库线程，get/post函数，后台新起的线程db

日志入库的消息队列本身使用的就是线程锁的机制

linux系统的协程phtread\_t,pthread\_create

c++11的线程std::thread

上述线程有的可以替换成协程交由调度器处理，待整理

##### 锁的替换

目前oss使用了三种锁待整理

* 公共库的锁CTMutex
* c++11的互斥量std::mutex
* linux的锁

pthread\_mutex\_t

pthread\_spinlock\_t

使用协程中的线程锁一定要换成协程锁。

### zk连接池

当前使用的是公共库的zk客户端，一个客户端代表一条连接

总体方案：

在初始化的时候，初始化多个zk链接（scheduler.xml设置个数）

当请求需要访问zk的时候，从zk链接池中拿去一条可用的链接进行操作

#### 背景

当前oss的状况是，只存在一个zk链接。由于oss的请求基本上都是操作zk（读取zk节点和写zk节点），整个程序运行过程中只有一个zk连接会产生大量的竞争（请求zk链接会加锁）。目前oss的性能瓶颈也是过去zk节点信息过慢导致满接口超时。因此基于协程框架下，实现一个zk连接池会显著提高oss的请求量qps。

代码中存在的问题：

由于只有一个zk客户端，在oss中使用的是全局变量的形式。包括以下三种形式：

tdsql\_zookeeper（share指针）

zk\_fd\_ (类中使用)

m\_zookeeper (类中使用)

三种使用全局zk客户端形式混用，导致改动量与难度增加了。

#### 方案

##### 增加zk连接池

// zk pool

class ZkPoolClient {

public :

ZkPoolClient(int n): count(0), sum(n) {}

int init() {}

boost::shared\_ptr<CConnectZk> get\_conn() {

// RR模式从链接池中取zk链接

return pool[(count++)%sum];

}

private:

// zk连接池

vector<boost::shared\_ptr<CConnectZk>> pool;

// 计数器

std::atomic<int> count;

// 链接总数

int sum;

};

配置文件新增zk链接链接数量，默认10

<zookeeper quit="1" iplist="{{zklist}}" timeout="10000" rootdir="{{rootdir}}" cluster="zk-test" spark\_rootdir="/noshard\_308" safeacl="0" conn\_num="10"/>

##### 使用宏定义全局变量

将原先代码中多处使用的全局变量zk客户端tdsql\_zookeeper定义为从zk连接池中取一条zk链接。

#define tdsql\_zookeeper oss\_zk\_pool->get\_conn()

每次调用全局zk时候，将从zk链接池中取一条zk链接使用，然后原子计数变量加一。

##### 慢接口优化

以上只是zk链接池的实现，争对具体的接口，还需要具体设计与优化。

这里慢接口主要存在两类：

* 使用面向过程流程写的GetGWGroup，GetGroup，GetInstance，QueryGWInstance四类查询接口
* 使用面向对象流程写的GetInsByIp，GetFenceInstance。

###### 两种接口区别分析

* 面向对象编写的接口(GetInsByIp)：

// 使用class resource\_set\_info的方法来从zk获取数据

ret = resource\_set\_info.get\_res\_set\_info(json\_object\_out["returnData"],

err\_msg);

// resource\_set\_info继承自job\_answer\_manager

// job\_answer\_manager则会在构造的时候默认使用tdsql\_zookeeper

// 则class resource\_set\_info将在构造初始话之后绑定一个zk链接

class job\_answer\_manager: public json\_operator{

public:

job\_answer\_manager(boost::shared\_ptr<CConnectZk> zk = tdsql\_zookeeper) {

zk\_fd\_ = zk;

rootdir\_ = zk\_fd\_->m\_rootdir;

zk\_iplist\_ = zk\_fd\_->m\_iplist;

}

~job\_answer\_manager() {}

}

* 面向过程编写的接口(GetGWGroup)：

// 大量使用tdsql\_zookeeper

// 这就导致了一个GetGWGroup请求将会使用多个zk链接，这也就引发了后续的"竞争"问题

ret = tdsql\_zookeeper->get\_gwset(setid, value, groupid);

###### 总结

面向对象编写的接口在调用的时候只会使用一条（或者不太多的zk链接）（只在类构造的时候选一条zk链接）

面向过程编写的接口在调用的时候会使用大量不同的zk链接（每次调用tdsql\_zookeeper都会从zk链接池中选取新的zk链接）

下面通过测试数据来分析，如何优化接口调用时间。

#### 以GetInsByIp为例

测试集群为广州集群，1400个实例

每秒100次接口调用，持续10s，

总计1000次请求，oss发起152w次zk调用

zk连接池使用默认10的连接数

在这之前，首先单个请求耗时是约为280ms

50、90、95、99 分别是攻击中所有请求延迟的第 50、90、95 和 99 个百分位数。

##### zk链接1 (即没有zk连接池)

Latencies [min, mean, 50, 90, 95, 99, max] 5.313s, 41.19s, 42.928s, 44.164s, 44.21s, 44.226s, 44.229s

平均耗时41s

##### zk连接数10

Latencies [min, mean, 50, 90, 95, 99, max] 992.296ms, 26.642s, 28.04s, 31.037s, 31.406s, 31.87s, 32.286s

平均耗时26s

GetInsByIp代码分析：

// 初始化对象，这里会在构造函数中给该对象分配一条zk链接

res\_set\_info resource\_set\_info;

ret = resource\_set\_info.get\_res\_set\_info(json\_object\_out["returnData"],

err\_msg);

* get\_res\_set\_info()

// get\_res\_set\_info()成员函数中有一个循环，会生成resource\_set\_info对象（该对象会在构造函数时，分配一条zk链接）。

for (auto port\_it = all\_port.begin(); port\_it != all\_port.end();

++port\_it) {

resource\_set\_info one\_set\_res(ip\_, \*port\_it, \*it);

Json::Value one\_set;

ret = one\_set\_res.get\_ressetinfo\_info(err\_msg);

if (ret == 0) {

if (!(it->empty())) {

one\_set["groupid"] = \*it;

para["groups"].append(one\_set);

break;

}

else {

one\_set["id"] = one\_set\_res.set\_id();

para["sets"].append(one\_set);

}

}

else {

runlog\_info("get the ressetinfo failed, the ret is %d", ret);

}

}

综上代码分析，GetInsByIp接口在一次请求中会循环使用不同的zk链接。也就是说，对于一次请求，其多次zk调用被分配到不同的zk链接去执行了。

当多个GetInsByIp接口被调用时，由于每次调用接口中的zk请求都被分散到了不同zk链接中，就导致一个接口的完成会线性依赖其他的zk链接的返回结果。

也就是说会产生一种竞争：所有的接口请求都在争抢所有的zk链接。

##### zk连接数10（一个请求对应一个链接）

结合上面的代码分析，这里做了点改动：一个GetInsByIp请求只会使用一条zk链接

// 在请求中初始化新的对象时，使用get\_zkfd()获取当前请求使用的zk链接，将其作为参数传给构造函数

// 这样新的对象将与原先使用统一个zk链接

resource\_set\_info one\_set\_res(ip\_, \*port\_it, \*it, get\_zkfd());

Latencies [min, mean, 50, 90, 95, 99, max] 479.372ms, 3.482s, 3.434s, 5.362s, 5.52s, 5.612s, 5.628s

平均耗时3.5s

这里可以看到耗时明显降低了，从41s到3.5s，耗时约为原先的十分之一。

#### 后续工作

对于面向过程编写的接口(GetGWGroup等)，进行面向对象编程的改造。

### 性能分析

#### 并发性能

采用集群规模：

40个group实例

每个group有2个set

每个set 为一主两备

初步估计一次GetGWGroup接口将要访问80-100次zk。

oss配置启动16线程

Latency :

min : 所有请求中最小的延迟

mean : 平均

50、90、95、99 分别是攻击中所有请求延迟的第 50、90、95 和 99 个百分位数。

max : 最大的延迟

##### 低并发 1s5次请求 qps=5

使用协程

Requests [total, rate] 50, 5.10

Duration [total, attack, wait] 10.755594967s, 9.79999987s, 955.595097ms

Latencies [mean, 50, 95, 99, max] 986.030528ms, 955.595097ms, 1.259408781s, 1.331306172s, 1.35530761s

Bytes In [total, mean] 6666850, 133337.00

Bytes Out [total, mean] 8800, 176.00

Success [ratio] 100.00%

Status Codes [code:count] 200:50

Error Set:

Latencies

mean : 986.030528ms

50, 95, 99 : 955.595097ms, 1.259408781s, 1.331306172s

max : 1.35530761s

使用线程

Requests [total, rate] 50, 5.10

Duration [total, attack, wait] 10.70469055s, 9.79999979s, 904.69076ms

Latencies [mean, 50, 95, 99, max] 854.453583ms, 826.60526ms, 1.089216839s, 1.113980973s, 1.115724104s

Bytes In [total, mean] 6665400, 133308.00

Bytes Out [total, mean] 8800, 176.00

Success [ratio] 100.00%

Status Codes [code:count] 200:50

Error Set:

Latencies

mean : 854.453583ms

50, 95, 99 : 826.60526ms, 1.089216839s, 1.113980973s

max : 1.115724104s

##### 高并发请求 1s50次请求

使用协程

Requests [total, rate] 500, 50.10

Duration [total, attack, wait] 44.685455224s, 9.979999786s, 34.705455438s

Latencies [mean, 50, 95, 99, max] 33.866959442s, 35.681655107s, 37.242088118s, 37.523833784s, 37.911612102s

Bytes In [total, mean] 66668500, 133337.00

Bytes Out [total, mean] 88000, 176.00

Success [ratio] 100.00%

Status Codes [code:count] 200:500

Error Set:

Latencies

mean : 33.866959442s

50, 95, 99 : 35.681655107s, 37.242088118s, 37.523833784s

max : 37.911612102s

使用线程

Requests [total, rate] 500, 50.10

Duration [total, attack, wait] 47.41425045s, 9.979999804s, 37.434250646s

Latencies [mean, 50, 95, 99, max] 19.387343784s, 19.546650084s, 35.260469051s, 36.905415921s, 37.438777846s

Bytes In [total, mean] 66654000, 133308.00

Bytes Out [total, mean] 88000, 176.00

Success [ratio] 100.00%

Status Codes [code:count] 200:500

Latencies

mean : 19.387343784s

50, 95, 99 : 19.546650084s, 35.260469051s, 36.905415921s

max : 37.438777846s

##### 分析

1、在低并发的情况下，协程和协程框架下，慢接口平均耗时相差无几

2、在高并发慢接口的情况下，协程的慢接口平均耗时大于线程。

3、协程下，所有的时间(mean,50,95,99,max)基本相同，符合协程的预期

原因：

oss目前只有一个zk链接，所有线程(协程都会争抢该锁)。但是由于线程得不到锁会hang住，但是对于协程，该线程并不会卡住，而会发生协程切换，并且切换的新的协程会继续争抢该锁。因此协程框架下，对该zk链接的抢用更为明显，次数更多。

#### 慢接口阻塞问题

经测试：

1、在线程框架下，当oss所有的线程都在处理慢接口时，此时oss无法处理任何请求（例如赤兔页面将无法刷新）

2、在协程框架下，即使oss所有线程都在处理大量慢接口，其余请求仍然能顺利执行（例如此时赤兔页面仍然能操作和刷新）

协程框架能解决慢接口阻塞导致oss完全无法服务的问题，解决其余快接口超时问题。

### 后续

经过分析，我们可以看到协程框架在oss现有框架下带来的优势，同时也存在不足（慢接口平均返回时间增加）。我们也可以得出，oss现在的对于慢接口的瓶颈在于整个oss只有一个公用的zk链接。

后续的改进方案：zk链接池改造

## SQLEngine支持bRPC

### 网络框架

1、在brpc中增加和mc 时间戳通道交互的协议TDGts protocol

2、将sqlengine和tdstore，mc的网络交互模式重构为brpc的channel模式

### 和前端mysql client的交互

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

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

### 锁，条件变量相关

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

### thread local

1、将current\_thd，THR\_MALLOC这种用于原本的线程级参数传递的变量修改为bthread local

2、自增值cache在和tdstore交互获取自增值的方面使用了thread local并且显示调用了libco的代码，整体重构为hash bucket+mutex的模式

3、open ssl中有thread local，mysql默认将open ssl的mutex注册为了mysql\_mutex xxx，而mysql mutex又被我们替换为了bthread mutex，这意味着open ssl内部使用mutex的时候可能产生切换，bthread被调度到其他pthread会导致thread local访问异常crash，需要取消mysql像open ssl注册修改mutex的行为

4、region cache使用了thread local，替换为hash bucket + mutex形式

5、storage/temptable下面的thread\_local uint8\_t \*shared\_block，thread\_local Tables tables按照bthread local改造

6、storage/perfschema/pfs.cc中的thread\_local PFS\_table\_context \*THR\_PFS\_contexts[THR\_PFS\_NUM\_KEYS]按照bthread local改造

7、sqlengine的百分位统计使用了thread local，考虑直接用bvar替换

### 对brpc channel的资源管理

1、mc域名对应的ip变更后需要重新初始化channel并更新到全局，这个过程中要保证和获取channel之间的线程安全并且不能有额外的性能损耗

2、tdstore ip变动后需要清理掉对应的无效channel

### 现有代码中的阻塞api以及显示的libco调用

1、目前代码中显示的poll调用会阻塞线程，需要替换为bthread可切换的模式，目前看这类调用并不多，多数为用于等待超市的功能，例如select sleep(5)，可以逐个手动修改

2、sqlengine互联使用了mysql client同步api，将server对fd和bthread的切换关联行为也应用在client lib中

3、并行框架中的libco调用需要改为bthread相关的语义

### 参数

1、变量持久化在mc后，需要处理grpc通道的channel初始化的参数获取问题（这是一类问题，包括—version的时候目前也会访问到没初始化好的参数）

2、和channel管理相关参数可配，方便性能调优，tdstore通道的连接数需要增加新的配置并且默认为1，brpc连接数过多容易性能下降

3、brpc中的负责处理工作线程和fd唤醒的epoll线程数改为可配，方便性能调优

4、网络相关的参数命名需要修改，也要通知云平做同步修改（这里需要考虑在之前的版本通过mc持久化的变量如果到了新的版本变量不存在了会导致sqlengine无法启动，是跳过还是升级的时候通过mc的接口先做修改再升sqlengine）

### 优化

1、不同类型的rpc设置独立的超时时间（比如get record超时应该比较短，count下推的rpc超时时间应该比较长）

2、通过sql查询bvar的统计信息/开发brpc的http端口查询bvar统计，方便性能分析

3、优化ebpf 获取rpc时耗的方式，降低ebpf trace的时候带来的性能损耗(已完成）

### 其他

1、sql执行过程中定期查看前端和client连接的socket是否正常，不正常需要设置对应thd的killed标志，防止sql在部分连接断开的情况下又”复活“修改数据的行为

2、清理和thread\_local mysql\_mutex\_lock\_num相关的调用

3、判断是否是后台线程的时候不能用thd->thread\_id,CurrentThdHelper中的set\_new\_thread\_id需要加回来，可以考虑用thd->is\_extra\_thread替代

4、调通unittest

5、确认iptable封网后brpc的恢复时间

6、升级brpc到1.0.0版本

7、移除libco相关的代码

### 性能

point select测试命令：

1024并发，12张表对应12个region分散到4个tdstore上，每张表200w行数据

./sysbench --time=60 --mysql-host=9.238.241.218 --mysql-port=8000 --mysql-db=test --tables=12 --table-size=2000000 --threads=1024 --mysql-user=test --mysql-password=test123 --report-interval=1 --auto\_inc=on --mysql-ignore-errors=1412,1180,2013,1205,1146,50018,1213,1049,50042 --percentile=99 --forced-shutdown=1 --create-secondary=0 src/lua/oltp\_point\_select.lua run

#### sqlengine配置80个工作线程

##### SQLEngine使用brpc改造前（libco模式）

###### sysbench统计

文本

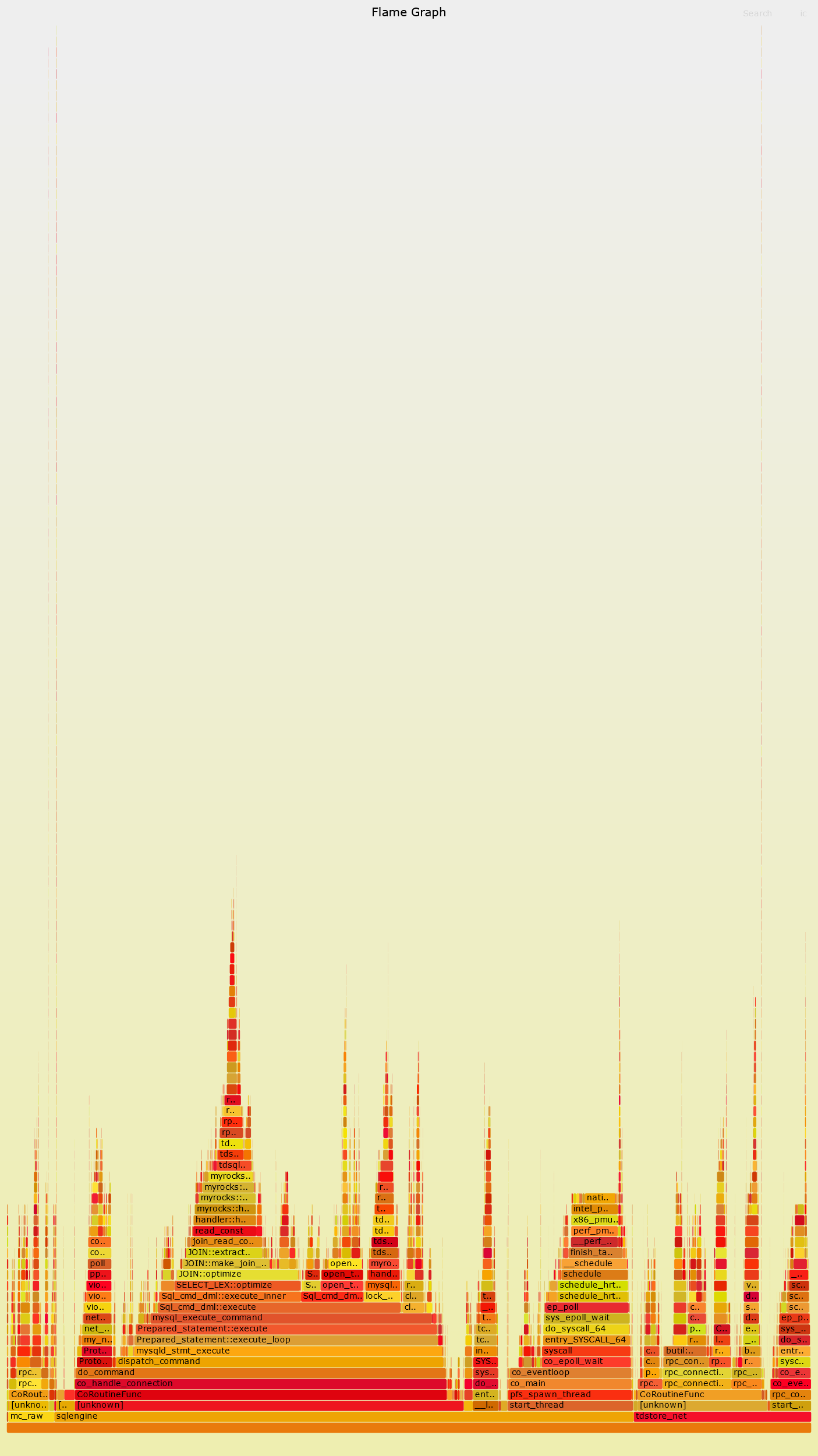
描述已自动生成

###### cpu消耗

图形用户界面

描述已自动生成

###### perf火焰图



城市的地标

描述已自动生成

##### SQLEngine使用brpc改造后

###### sysbench统计

文本

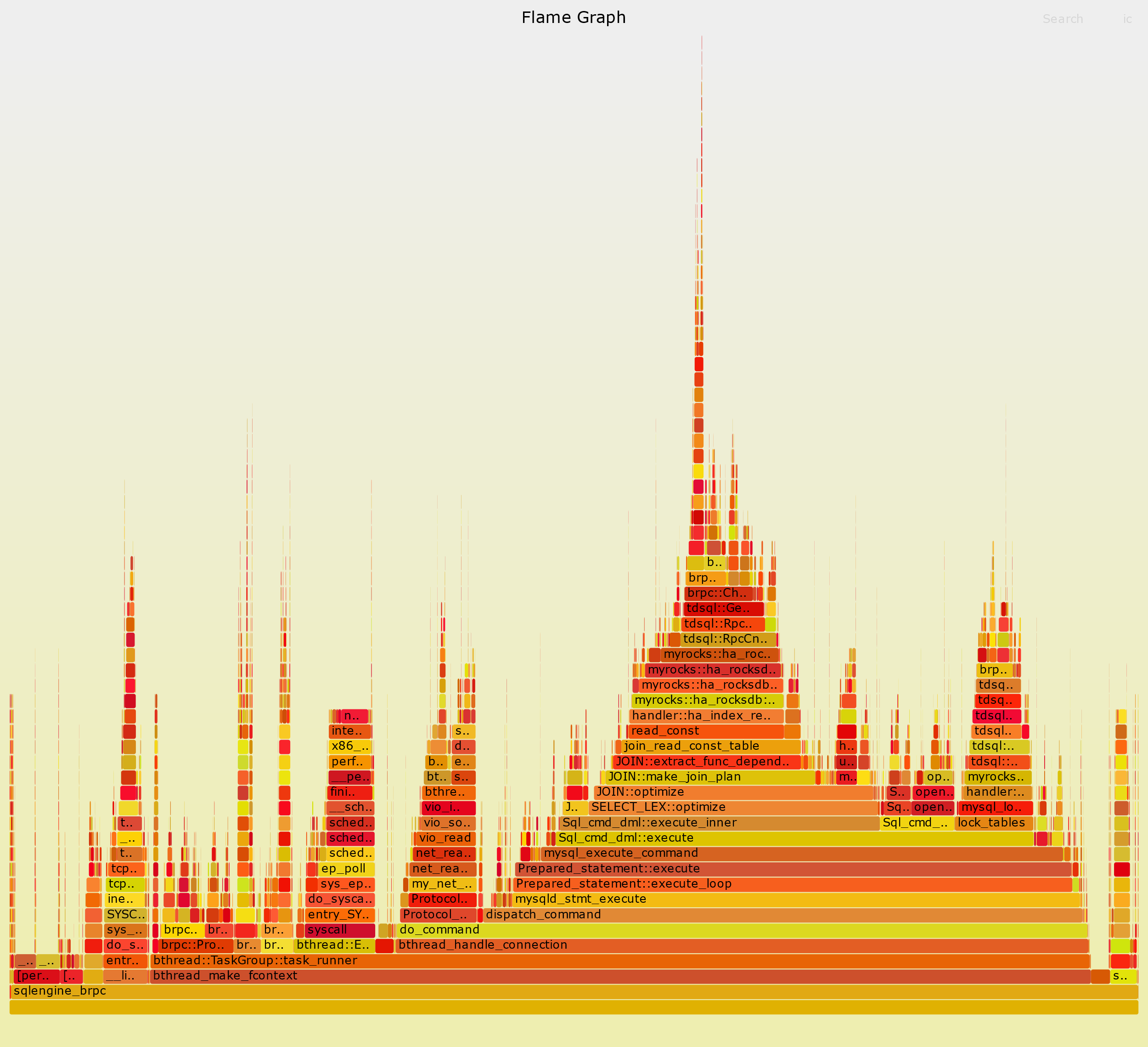
描述已自动生成

###### cpu消耗

图形用户界面

中度可信度描述已自动生成

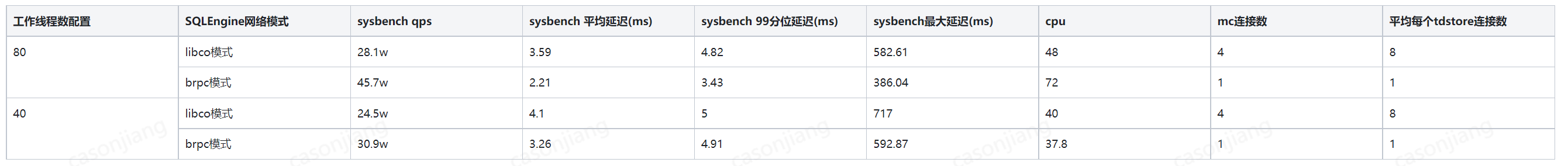
###### perf火焰图



城市的地标

描述已自动生成

#### 数据汇总



## TDSQL3.0支持协程

### 协程交互逻辑

基本流程：

1. SQLEngine启动时参数thread\_handling=co-threads，系统初始化协程环境
2. SQLEngine以协程模式处理用户请求

具体如下：

//sqlengine 会和一个mc，一个mc\_raw以及N个tdstore交互

协程模式有两种，由参数rocksdb\_tdstore\_per\_thread和rocksdb\_mc\_per\_thread控制。

1、当为1的时候，处理逻辑简单，但是连接不收敛，一个客户端连接对应多个后端连接（1+1+N)，如果有1000个客户端连接的话，和后端的连接时1000\*（1+1+N）

2、当为2的时候，处理逻辑稍微复杂些，连接可以收敛，不管多少个客户端连接，和后端的连接都是rocksdb\_net\_threads\_num\*(1+1+N)

//正式环境用模式2，模式1主要为了必要的时候做对比测试

处理流程如下：

//初始化的时候会建立co\_threads\_num个工作线程，（模式2的时候会额外建立rocksdb\_net\_threads\_num网络线程）

1、每接收一个连接，会新启一个会话co负责处理，这个co会对应一个thd结构，这个co会一直跑在某个工作线程上

2、当会话co需要向前段发送接收数据的时候，由于每个co对应一个fd，不用额外处理

3、当会话co需要向后端（tdstore或者mc）发送和接收数据的时候，统一通过这三个函数进行数据交互，call\_mc\_rpc\_co,call\_tdstore\_rpc\_co,call\_mc\_raw\_co ,不过具体根据模式不同处理逻辑不一样

//两种模式的区别在于，是直接唤醒工作线程上的extra\_write\_co，

//还是通过eventfd通知网络线程上的extra\_co，让它再去唤醒网络线程上的extra\_write\_co

1）模式1:call\_mc\_rpc\_co\_work\_thread,call\_tdstore\_rpc\_co\_work\_thread,call\_mc\_raw\_co\_work\_thread

//id用于区分thd和回包的关系

thd->id = tdstore\_correlation\_id[index]++;

//要发送的数据写入thd->w\_buf中

thd->w\_buf.append(::butil::IOBuf::Movable(request\_buf));

tdstore\_socket\_co->thds.push\_back(thd);

//唤醒本线程中负责发送数据的协程extra\_write\_co

co\_cond\_signal(tdstore\_socket\_co->co\_socket\_cond);

2）模式2:call\_mc\_rpc\_co\_net\_thread,call\_tdstore\_rpc\_co\_net\_thread, call\_mc\_raw\_co\_net\_thread

//确定该会话co由哪个网络线程负责

int index = thd->thread\_id() % co\_threads\_num;

index = index % rocksdb\_net\_threads\_num;;

thd->rpc\_type=RPC\_TDSTORE;

//id用于区分thd和回包的关系

thd->id = tdstore\_correlation\_id[index]++;

//要发送的数据写入thd->w\_buf中,把thd结构传给网络线程，并且通知它上面的extra\_co

thd->w\_buf.append(::butil::IOBuf::Movable(request\_buf));

queue\_request[index].push(thd);

uint64\_t s = write(request\_fds[index], &u, sizeof(uint64\_t));

//extra\_co,每个net线程一个

//监听request\_fds,获取需要发送数据的所有thd结构

s = read(request\_fds[index], &u, sizeof(uint64\_t));

while

queue\_request[index].pop(thd);

//根据类型，对thd进行分类，存入数据结构分量thds，并且通知真正读写数据的extra\_write\_co

tdstore\_socket\_co\_maps[index][thds[i]->tdstore\_conn\_id].thds.push\_back(thds[i]);

co\_cond\_signal(tdstore\_socket\_co.second.co\_socket\_cond);

3）extra\_write\_co,每个工作（模式1）/网络线程（模式2）----1+1+N

负责建立真正的连接

for

//把所有的数据合成一个

request\_buf.append(::butil::IOBuf::Movable(co\_socket->thds[i]->w\_buf));

co\_socket->thds[i]->fd\_version=co\_socket->fd\_version;

co\_socket->thds[i]->send\_time=time(NULL);

co\_socket->co\_thds[co\_socket->thds[i]->id] = co\_socket->thds[i];

int ret = write\_iobuf(fd, request\_buf);

co\_thds数组中记录了该write co负责的所有thd的信息，后续根据接收到的包的id进行唤醒操作

### 协程锁同步

#### 背景

SQLEngine基于原生mysql改造，而mysql的并发是基于线程模型的，不管是per-thread，还是thread-pool，正在执行sql的连接每个都独占一个线程，通过pthread相关的同步原语进行并发控制和等待。

为了性能，SQLEngine同时做了协程改造（基于libco），每个连接对应一个协程，不再独占一个线程，但是同步相关的函数还是使用pthread，存在死锁风险

死锁例子：

co1 co2

lock

read

lock

如果co1和co2对应两个连接的协程，在一个线程中，那么当co1执行lock后，再执行read的时候会进行协程切换，此时co2开始执行，当其执行lock的时候会一直阻塞整个线程，co1没有机会再次执行，系统死锁

#### 方案设计

##### 当前解决方案

当co2执行lock的时候会判断该锁是否是当前线程获取，如果是的话，则退成协程的等待机制（libco提供了同一个线程内的等待唤醒机制）

存在的问题：

1、对业务层的侵入比较大

2、对于mutex，可以保证不死锁，但是会有一个协程阻塞其他协程的问题

3、不适用于读写锁，不适合条件变量

##### 解决方案

要彻底解决有两条路径

1、基于brpc实现mysql协议的编解码，这一块对现有逻辑改动会比较大，同时为了后续的问题解决，对用brpc内部的bthread，butex等相关的细节也需要完全掌握

2、在libco在协程库层实现跨线程的锁功能，对业务层屏蔽具体的逻辑。新的libco提供了一些协程锁相关的逻辑，但是目前没有hook pthread相关函数的计划，因此需要和原生mysql做适配，同时这块代码比较新，却又非常关键，对于可靠性要求很高

目前看第二种方案对现有架构改动比较小，唯一的风险是代码的可靠性，需要做比较多的测试

#### 概要设计

1、当前mysql使用的同步功能：

普通mutex：mysql\_mutex\_lock，

读写锁：mysql\_rwlock\_rdlock，mysql\_rwlock\_wrlock

读优先的读写锁：mysql\_prlock\_rdlock， mysql\_rwlock\_wrlock

条件变量：mysql\_cond\_wait

其最底层利用的pthread提供的相关数据结构和函数

pthread\_mutex\_t，pthread\_rwlock\_t，pthread\_cond\_t

2、libco目前提供的同步功能（http://git.code.oa.com/libco/colib.git）

detail::LockEntry：基于eventfd实现了跨线程的等待和唤醒功能

libco::ConditionVariable：基于detail::LockEntry和std::mutex实现的条件变量功能

libco:: CoMutex：基于libco::ConditionVariable和std::mutex实现的mutex功能

libco:: CoRWMutex：基于libco::ConditionVariable和std::mutex实现的读写mutex功能

目前发现mutex的trylock实现有问题，已经和wx那边确认修复

同时rwmutex实现有问题，无法使用

##### 协程锁同步总体设计

在现有基础上需要做两部分的改动

1、mysql的原生代码在尽量少改动的情况下，调用libco提供的同步功能

2、libco需要适配改造适应pthread的相关使用方式

###### mysql代码改造

原生代码：

static inline int native\_mutex\_lock(native\_mutex\_t \*mutex) {

#ifdef \_WIN32

EnterCriticalSection(mutex);

return 0;

#else

return pthread\_mutex\_lock(mutex);

#endif

}

通过编译选项控制底层实现逻辑，以mutex为例，部分代码如下：

#ifdef CO\_MUTEX

typedef libco::CoMutex native\_mutex\_t;

typedef int native\_mutexattr\_t;

#else

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

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

#endif

static inline int native\_mutex\_init(native\_mutex\_t \*mutex,

const native\_mutexattr\_t \*attr ) {

#ifdef CO\_MUTEX

return 0;

#else

return pthread\_mutex\_init(mutex, attr);

#endif

}

static inline int native\_mutex\_lock(native\_mutex\_t \*mutex) {

#ifdef CO\_MUTEX

mutex->lock();

return 0;

#else

return pthread\_mutex\_lock(mutex);

#endif

}

static inline int native\_mutex\_unlock(native\_mutex\_t \*mutex) {

#ifdef CO\_MUTEX

mutex->unlock();

return 0;

#else

return pthread\_mutex\_unlock(mutex);

#endif

}

###### libco代码改造

libco的mutex是基于它对应的条件变量实现的，而pthread的条件变量依赖于它对应的mutex：

int pthread\_cond\_wait(pthread\_cond\_t \*restrict cond, pthread\_mutex\_t \*restrict mutex);

libco要想提供和pthread一样的使用逻辑的话，需要提供依赖于mutex的条件变量，而不是相反

因此对于ConditionVariable做模板化改造：

ConditionVariable< std::unique\_lock<std::mutex> > //内部使用

libco:: CoMutex //提供给业务的mutex

libco::ConditionVariable< libco:: CoMutex> //提供给业务的条件变量

ConditionVariable能否提供和pthread\_cond\_wait完全一致的原子性？

Libco中实现的mutex以类为基础，但是mysql对于mutex的使用很随意，不能有类的构造函数，因此内部使用std::mutex和std::atomic都必须重新改写

## 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。