# 概述

# 原理

# 应用

## TDSQL3.0后台MySQL连接

### 背景

在TDSQL SQLEngine中

### 方案

## TDSQL3.0 SQLEngine RPC改造

### 改造方案

1、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

2、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

3、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

4、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

5、对brpc channel的资源管理

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

tdstore ip变动后需要清理掉对应的无效channel

6、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

Sqlengine执行器与Rpc性能优化：

一些Rpc是串行发送的，比如Scan是依次在每个Region上扫描的，如果可以做并行的Scan那么工作协程可以减少等待网络的时间。



### 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

#### 测试命令

针对点查（point select）的测试命令：

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s

./sysbench --time=60 --mysql-host=9.x.x.x --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

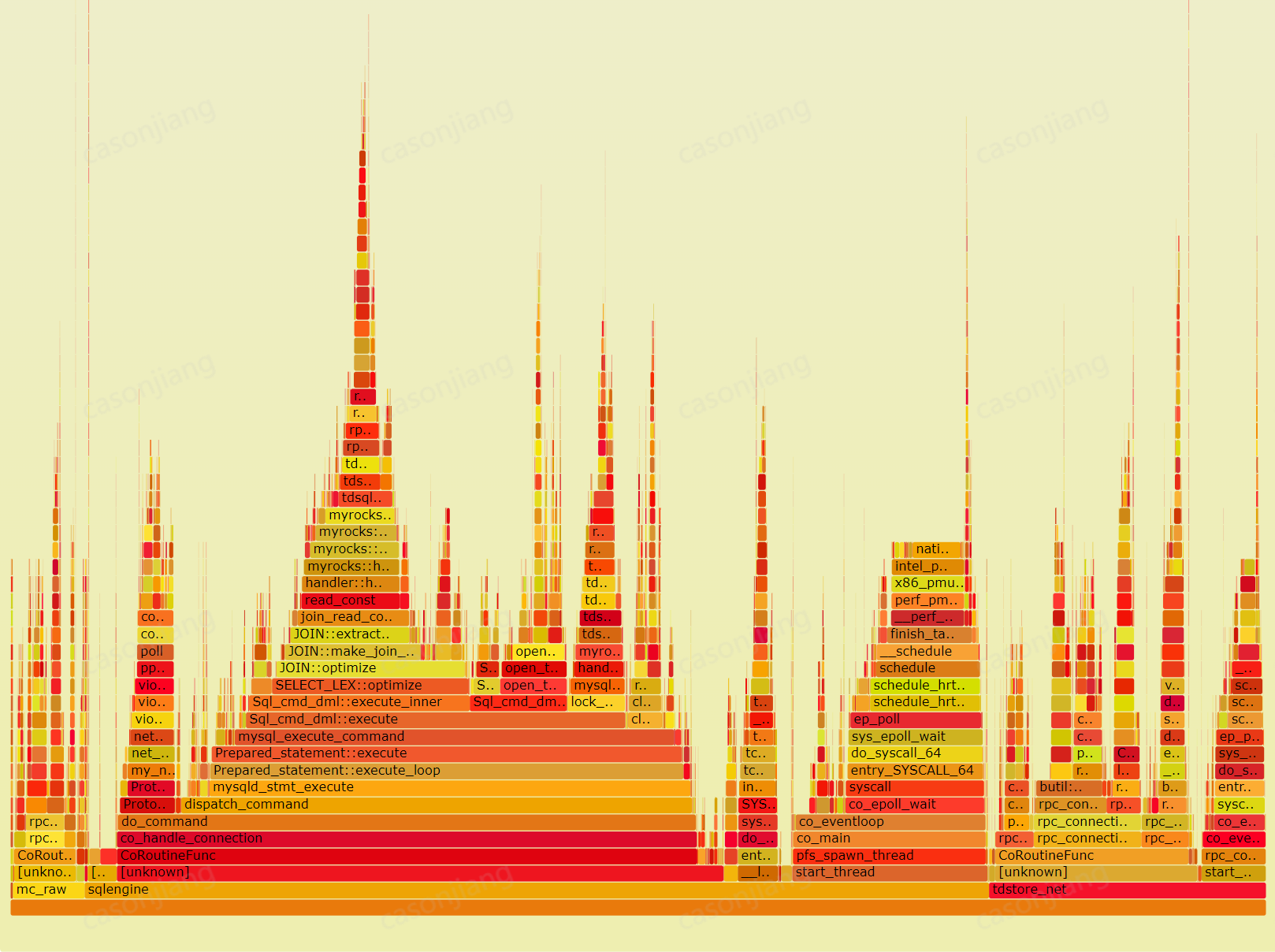
描述已自动生成

###### cpu消耗

图形用户界面

描述已自动生成

###### Perf火焰图



##### SQLEngine使用brpc改造后

###### sysbench统计

文本

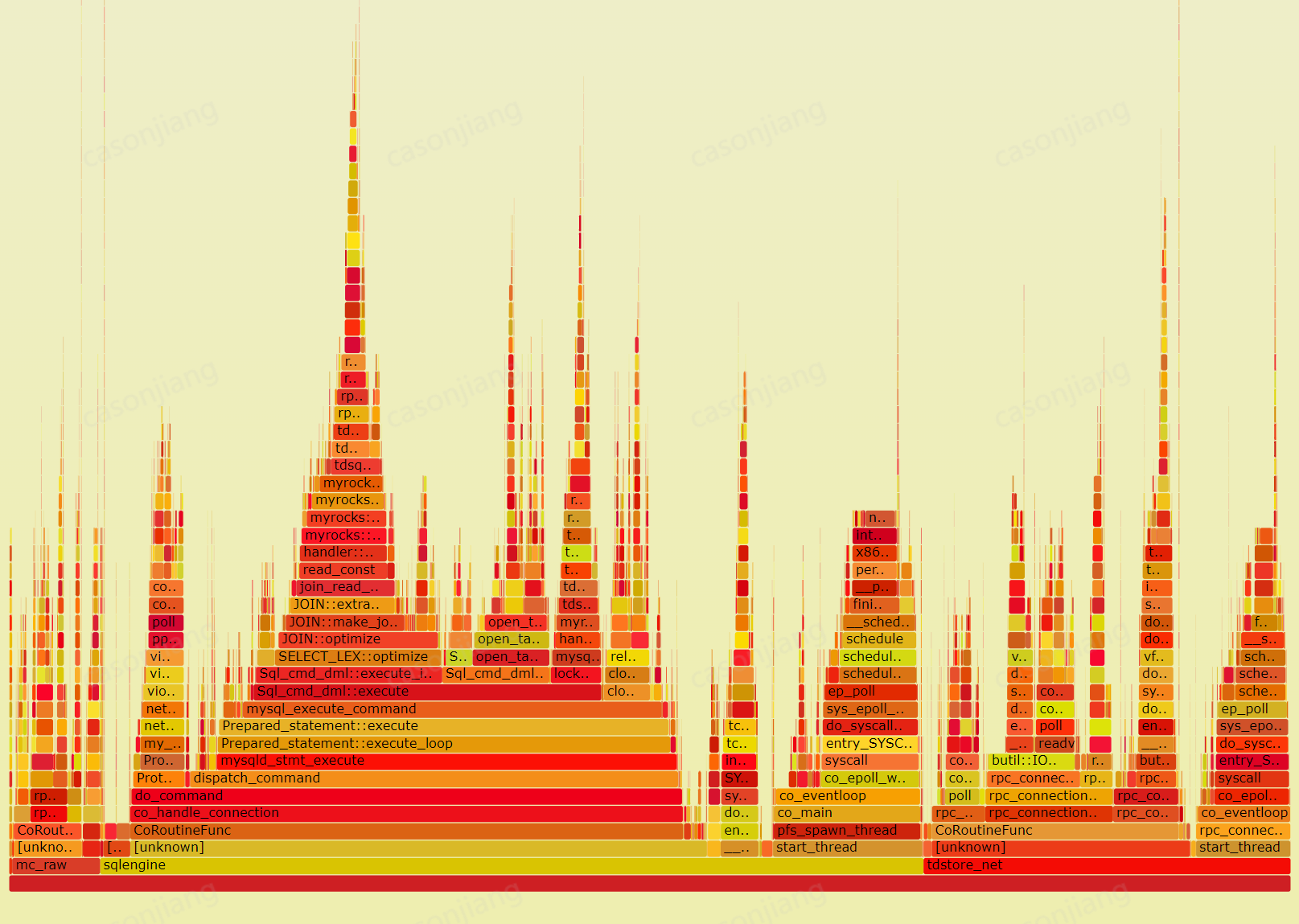
描述已自动生成

###### CPU消耗

图片包含 文本

描述已自动生成

###### Perf火焰图



## TDSQL2.0 OSS协程改造

### 背景

oss在迭代22.1中为了使用协程，引入了brpc。

该brpc源码以submodule的形式引入oss源码中。

1. 这里考虑到以cmake形式引入的brpc编译具有可复用性质（其他模块可以参照这个模版引入brpc）。
2. 其次对于brpc中的bthread的单独使用，官方并没有出专门的编写文档（官方bthread介绍）。因此这里总结下bthread的使用案例。

### 编译

引入BRPC编译模版：

理论上完全按照oss方式的目录配置cmake文件与目录，可以直接完整的编译出brpc库。

#### git submodule

[submodule "brpc"]

path = extra/incubator-brpc/incubator-brpc-1.0.0

url = https://github.com/apache/brpc.git

branch = master

#### 目录划分

cmake目录

├── gflags.cmake

├── glog.cmake

├── incubator\_brpc.cmake

├── install\_layout.cmake

├── install\_macros.cmake

├── leveldb.cmake

├── libutils.cmake

├── protobuf.cmake

├── protobuf\_proto\_compile.cmake

└── snappy.cmake

extra目录

├── CMakeLists.txt

├── gflags

├── glog

├── incubator-brpc

├── leveldb

├── protobuf

└── snappy

extrea：brpc源码与brpc编译依赖模块的源码目录。除了incubator-brpc是通过git submodule引入的。

cmake：cmake目录中的文件为编译brpc与brpc依赖的各种参数与环境配置

#### 编译

1、为了方便单独编译brpc模块，可以使用cmake中的add\_subdirectory方式来单独编译

# 编译brpc

add\_subdirectory(${PROJECT\_SOURCE\_DIR}/extra)

2、include brpc的头文件目录

include\_directories(

# brpc

${brpcdir}/output/include

${project\_root}/extra/incubator-brpc/incubator-brpc-1.0.0/src

${gflagsdir}/include

)

3、链接brpc静态库

add\_dependencies(${PROJECT\_NAME} brpc-static)

target\_link\_libraries(${PROJECT\_NAME} brpc-static)

### bthread的使用

（官方bthread介绍）

包含头文件#include "bthread/bthread.h"

#### brpc的互斥量与条件变量

由于brpc中引入了bthread，如果在bthread中使用了mutex，那么将会挂起当前pthread，导致该bthread\_worker无法执行其他bthread，因此类似pthread和futex的关系，brpc引入butex来实现bthread粒度的挂起和唤醒。

##### unix格式的butex使用

typedef struct {

unsigned\* butex;

bthread\_contention\_site\_t csite;

} bthread\_mutex\_t;

// init mutex

extern int bthread\_mutex\_init(bthread\_mutex\_t\* \_\_restrict mutex,

const bthread\_mutexattr\_t\* \_\_restrict mutex\_attr);

// Destroy `mutex'.

extern int bthread\_mutex\_destroy(bthread\_mutex\_t\* mutex);

// Wait until lock for `mutex' becomes available and lock it.

extern int bthread\_mutex\_lock(bthread\_mutex\_t\* mutex);

// Unlock `mutex'.

extern int bthread\_mutex\_unlock(bthread\_mutex\_t\* mutex);

例子

bthread\_mutex\_t mutex;

void\* thread\_func(void\* arg) {

bthread\_mutex\_lock(&mutex);

// 临界区

bthread\_mutex\_unlock(&mutex);

return NULL;

}

int main() {

bthread\_mutex\_init(&mutex, NULL);

bthread\_t tid;

bthread\_start(&tid, NULL, thread\_func, NULL);

// ...

bthread\_mutex\_lock(&mutex);

// 临界区

bthread\_mutex\_unlock(&mutex);

// ...

bthread\_join(tid, NULL);

bthread\_mutex\_destroy(&mutex);

return 0;

}

##### c++11格式的butex使用

class Mutex {

public:

Mutex() {

int ec = bthread\_mutex\_init(&\_mutex, NULL);

if (ec != 0) {

throw std::system\_error(std::error\_code(ec, std::system\_category()), "Mutex constructor failed");

}

}

~Mutex() { CHECK\_EQ(0, bthread\_mutex\_destroy(&\_mutex)); }

void lock();

void unlock() { bthread\_mutex\_unlock(&\_mutex); }

bool try\_lock() { return !bthread\_mutex\_trylock(&\_mutex); }

private:

DISALLOW\_COPY\_AND\_ASSIGN(Mutex);

bthread\_mutex\_t \_mutex;

};

bthread::Mutex本质上就是对bthread\_mutex\_t的封装类。与c++11中std::mutex的使用方法类似。

例子：

# 初始化与上锁

bhtread::Mutex lock\_;

lock\_.lock();

lock\_.unlock();

# 使用std::lock\_guard来上锁

std::lock\_guard<bthread::Mutex> lock(lock\_);

bthread中条件变量的使用

bthread::ConditionVariable cond\_;

bthread::Mutex m\_lock;

// 等待

{

std::unique\_lock<bthread::Mutex> lock(m\_lock);

cond\_.wait(lock);

}

// 唤醒

{

std::unique\_lock<bthread::Mutex> lock(m\_lock);

cond\_.notify\_one();

}

#### bthread的使用

##### 设置bthread使用的线程数量

// Set number of worker pthreads to `num'. After a successful call,

// bthread\_getconcurrency() shall return new set number, but workers may

// take some time to quit or create.

// NOTE: currently concurrency cannot be reduced after any bthread created.

extern int bthread\_setconcurrency(int num);

##### bthread的启动

作为一个线程库，用户最常用的自然是启动线程，bthread的启动入口函数有两个：

1、bthread\_start\_urgent：让出当前worker立即执行新bthread。

2、bthread\_start\_background：将要启动的bthread放入队列等待调度。

// Create bthread `fn(args)' with attributes `attr' and put the identifier into

// `tid'. Switch to the new thread and schedule old thread to run. Use this

// function when the new thread is more urgent.

// Returns 0 on success, errno otherwise.

extern int bthread\_start\_urgent(bthread\_t\* \_\_restrict tid,

const bthread\_attr\_t\* \_\_restrict attr,

void \* (\*fn)(void\*),

void\* \_\_restrict args);

// Create bthread `fn(args)' with attributes `attr' and put the identifier into

// `tid'. This function behaves closer to pthread\_create: after scheduling the

// new thread to run, it returns. In another word, the new thread may take

// longer time than bthread\_start\_urgent() to run.

// Return 0 on success, errno otherwise.

extern int bthread\_start\_background(bthread\_t\* \_\_restrict tid,

const bthread\_attr\_t\* \_\_restrict attr,

void \* (\*fn)(void\*),

void\* \_\_restrict args);

##### 等待bthread结束

// Make calling thread wait for termination of bthread `bt'. Return immediately

// if `bt' is already terminated.

// Notes:

// - All bthreads are "detached" but still joinable.

// - \*bthread\_return is always set to null. If you need to return value

// from a bthread, pass the value via the `args' created the bthread.

// - bthread\_join() is not affected by bthread\_interrupt.

// Returns 0 on success, errno otherwise.

extern int bthread\_join(bthread\_t bt, void\*\* bthread\_return);

##### bthread专用的sleep函数

bthread中使用sleep()和usleep()函数将会导致pthread挂起，因此要使用bthread的专用sleep函数来保证bthread的正常切换。

// Suspend current thread for at least `microseconds'

// Interruptible by bthread\_interrupt().

extern int bthread\_usleep(uint64\_t microseconds);

说明：这个要格外注意，因为有时候可能会对这类睡眠函数进行封装，比如对外呈现的都是SLEEP(N)，但是底层实现需要对pthread和bthread区分。

##### 使用例子

// 设置启动pthread数，最小为4

bthread\_setconcurrency(4);

// 启动一个bthread

bthread\_t bid;

std::string output = "test case";

// 这里我们使用lamuda函数进行传参

bthread\_start\_background(&amp;res\_tid, NULL, [](void\* arg)->void\* {

std::string tmp = \*static\_cast<std::string\*>(arg);

std::cout << tmp << std::endl;

return nullptr;

}, &amp;output);

// 等待bthread结束

bthread\_join(bid, nullptr);