Libco 分享

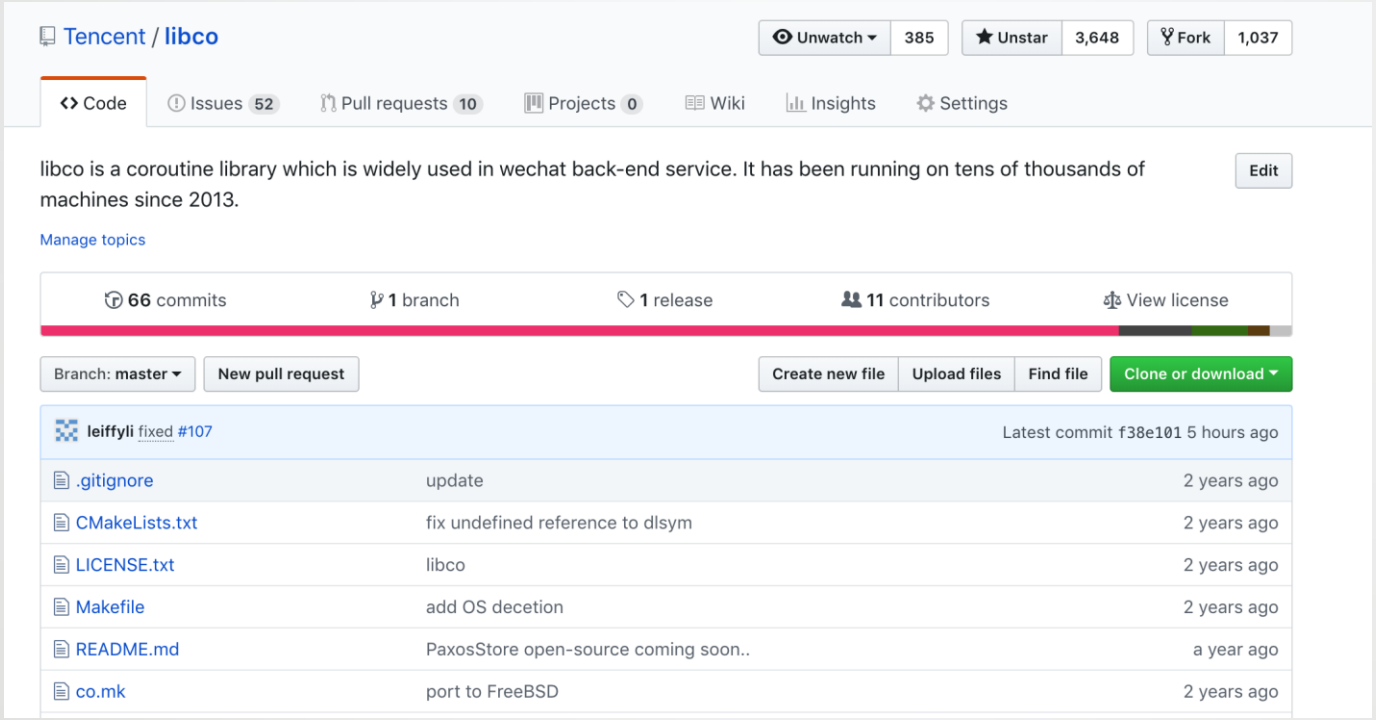
腾讯 – 李方源（Leiffyli)

自我介绍

- Leiffyli， 李方源，微信后台高级工程师；
- 2013年研究生毕业加入腾讯；
- Libco开源项目负责人；
- 先后参与微信后台协程化改造（Libco）、微信后台RPC框架重构与开发等项目；
- 目前负责微信后台基础服务框架、基础组件的设计及维护；

Libco开源现状

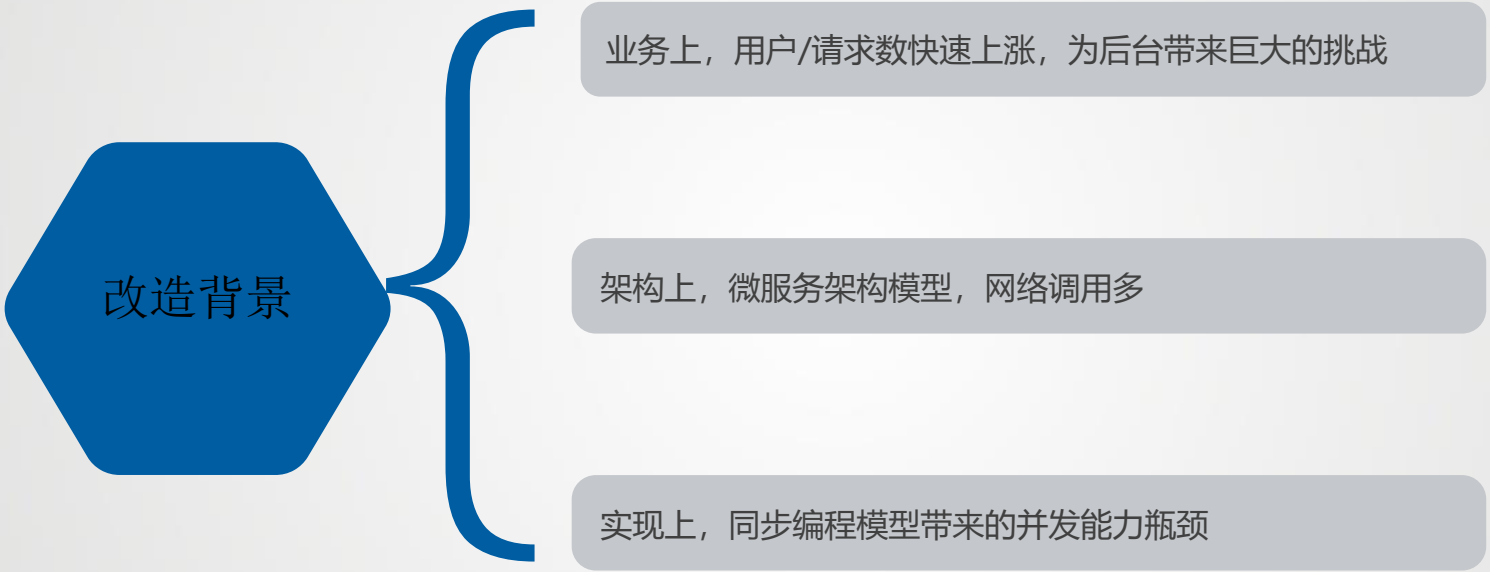
- 开源地址：https://github.com/Tencent/libco
- 代码：核心文件3个，代码2k+行



微信Libco线上现状

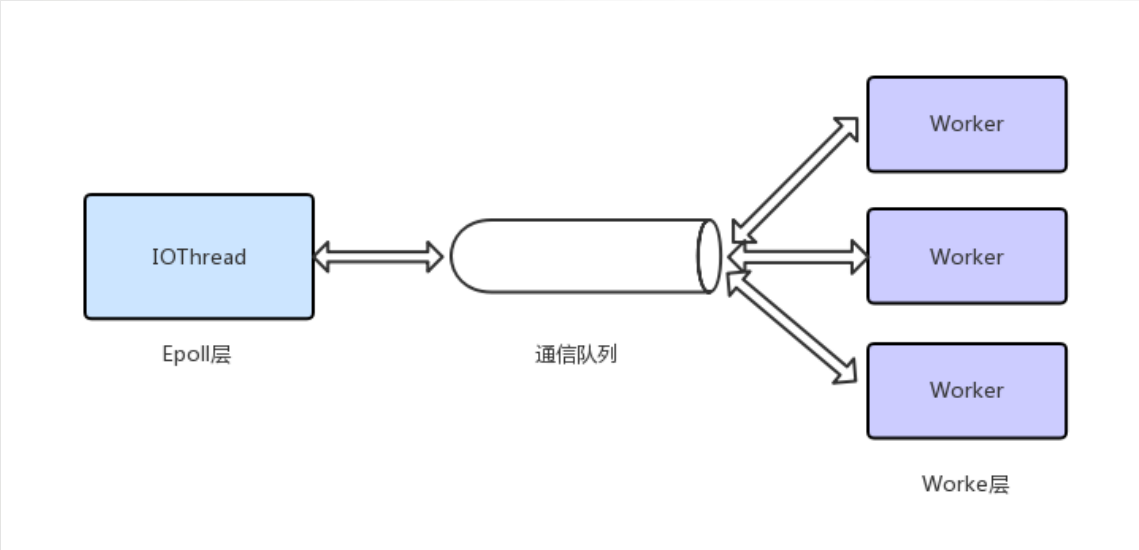
- 运行在微信**绝大部分服务**、数万台机器上；
- 除了明确的计算行服务，或者无任何IO操作的服务不开启协程，其它服务都**默认使用协程**；
- 13年改造至今运行稳定；

Libco产生背景

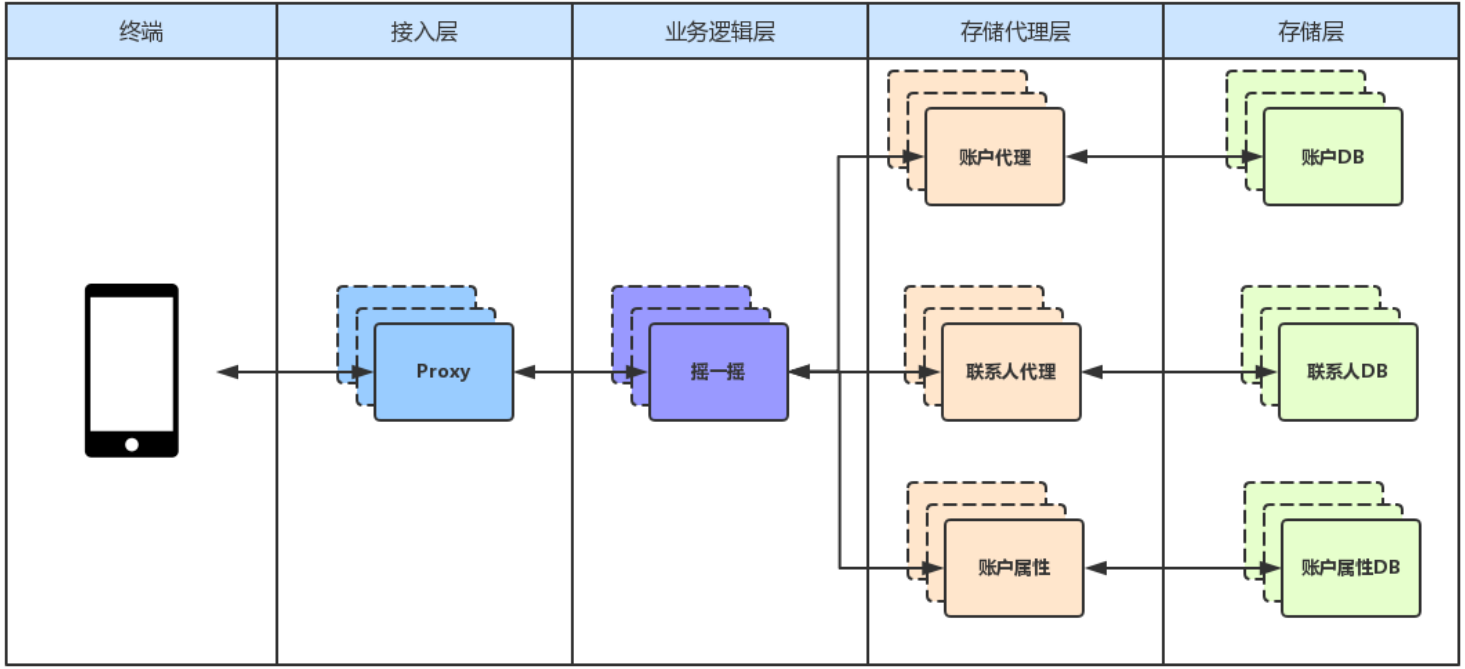

改造背景
业务上，用户/请求数快速上涨，为后台带来巨大的挑战
架构上，微服务架构模型，网络调用多
实现上，同步编程模型带来的并发能力瓶颈

微信的RPC框架

- 2013年协程改造前的微信RPC框架
  - 半同步半异步模型
  - 接入层Epoll层为纯异步模型，负责网络收发包
  - Worker层为同步模型，负责业务逻辑处理，一般配置为几十到几百；



微信的微服务架构


终端
接入层
业务逻辑层
存储代理层
存储层
Proxy
摇一摇
账户代理
联系人代理
账户属性
账户DB
联系人DB
账户属性DB

迫切需求 - 提升系统的并发能力

- 系统中业务逻辑涉及的网络IO操作均为同步操作；

```
int SendMessage(const Req& req, const Resp& resp) {
    req.SerializeToBuffer(&send_buffer;)   //序列化到buffer
    SetFdTimeout(fd, conn_timeout, sock_timeout): //设置超时时间

    //网络调用
    Connect(fd, ip, port) //发起连接;
    SendRequest(fd, &send_buffer): //发送请求包;
    RecvResponse(fd, &recv_buffer); //等待回包;

    resp.ParseFromBuffer(recv_buffer):   //反序列化
}
```

- 异步化改造
  - 纯异步模型改造
    - 模块数：数百
    - 代码行数： 百万级

平稳改造几乎是不可能完成的任务

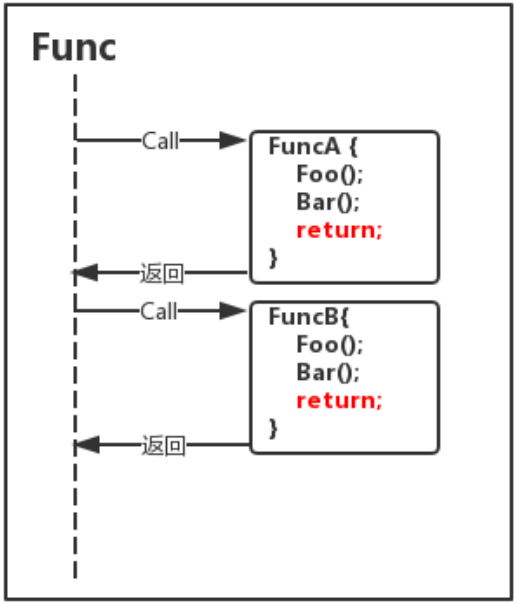
寻求其它改造方法

- 协程

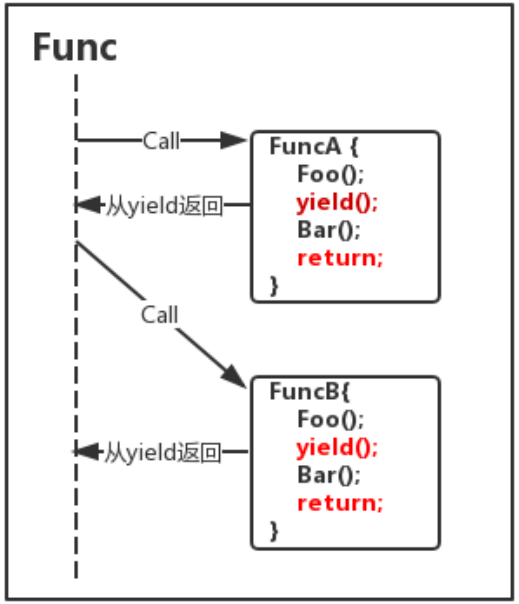
什么是协程

- 微线程，用户态调度的协程

普通函数调用



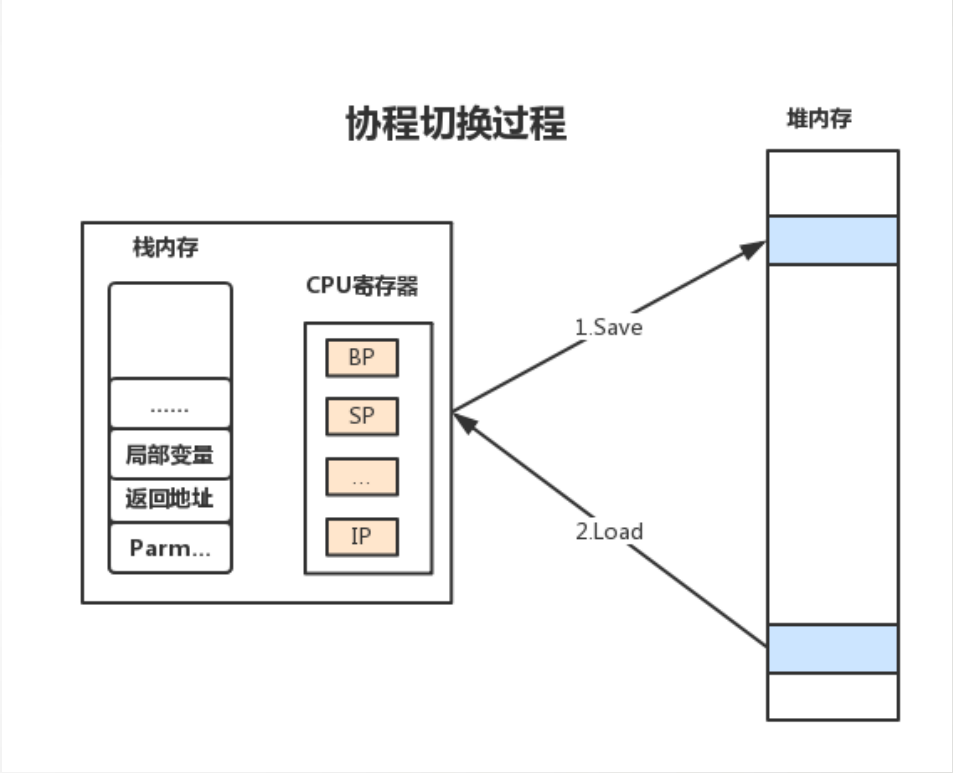
协程函数调用



怎么做协程切换

- 协程切换：保存当前函数的执行状态，导出目标函数上一次退出运行时的状态

- 函数执行状态包括：
  - cpu寄存器
  - 当前函数执行栈内容



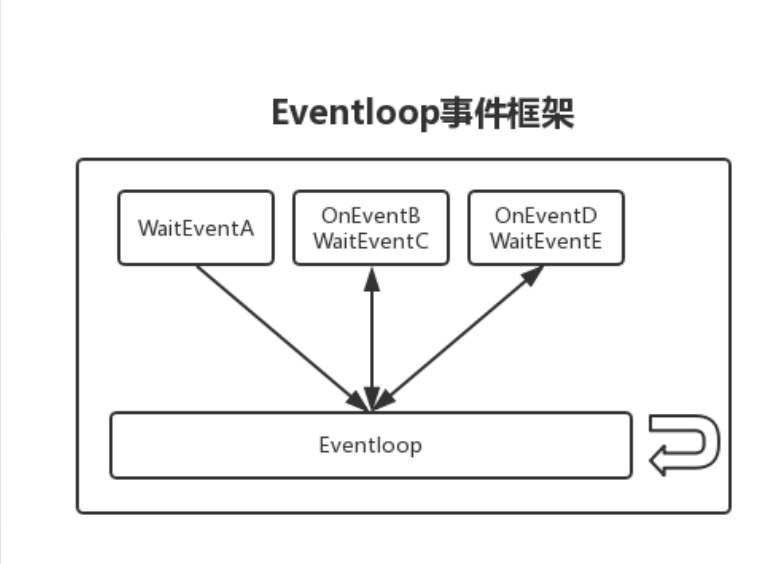
栈内容是否要保存

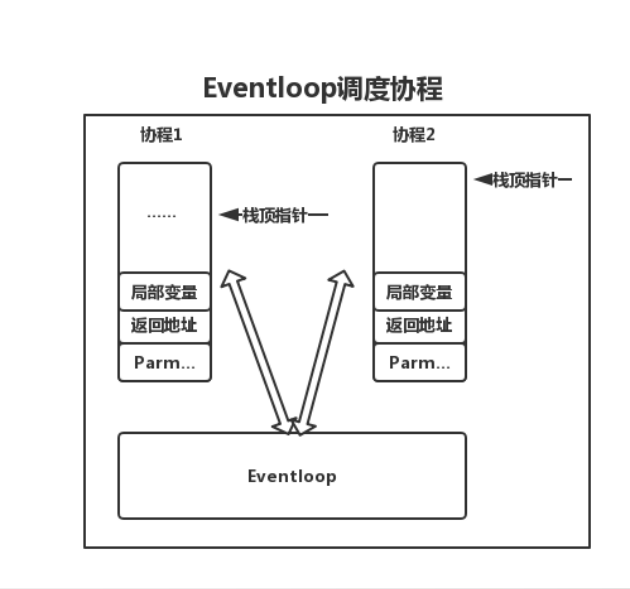
- **共享栈**：协程切换会保存栈内容，多个协程共用同一片内存空间
  - 优点
    - 协程使用的栈空间可以开的比较大
  - 缺点
    - 每次拷贝需要耗费额外cpu
    - 栈地址不可跨协程使用；

- **私有栈**：每个协程独立运行在自己的栈内存空间中
  - 优点：
    - 不需拷贝栈内存，性能高；
    - 独占栈地址，使用安全；
  - 缺点：
    - 可能会占用较多内存（操作系统对大内存一般只分配地址空间，真实使用时才会触发缺页中断，申请物理内存）；

协程的最佳调度时机

- 解决cpu利用率与IO利用率不平衡的问题
  - 阻塞等待IO会导致CPU资源浪费；
- 事件驱动（eventloop)框架





同步框架改造协程的方案选择

1. **显式协程切换**
   - 找出系统中所有同步阻塞调用，显式改成协程切换

2. **自动协程切换（Hook Socket族方法）**
   - 切换时机为同步网络调用时候
     - 网络未准备的时候让出
     - 网络事件发生或超时的时候恢复

```
// Hook Read系统调用伪代码;
function Read(fd, buf, size) {
    if (fd not ready) {
            AddEvent(fd, EPOLLIN, timeout); //添加事件到事件循环
            Yield(); //让出协程控制权
            //协程重新获得控制权，将从下一句指令开始执行;
    }
    return SysRead(fd, buf, size); //调用系统Read调用

}
```

Libco架构

基于汇编实现的协程切换内核
裁剪了部分现网实现上不需保存的寄存器；
尽可能的让协程切换高效率；

协程源语
封装了协程必须的几个源语api；
co_create/ co_resume/ co_yield等

Socket族函数Hook
Hook了大部分Socket族相关的api，使得协程切换时机自动与网络事件或超时绑定；

轻量网络框架
Epoll/Kqueue事件驱动+时间轮盘超时管理，组成一个轻量级异步网络框架；

Libco的协程实现

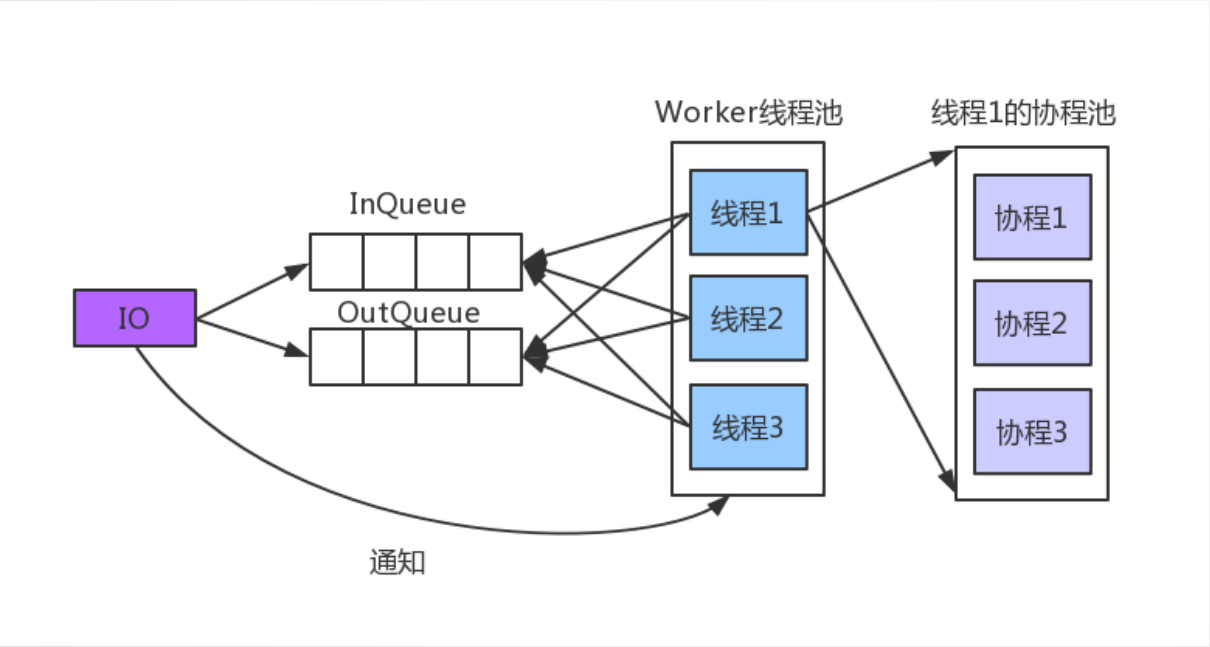
- 协程切换实现
  - 精简的汇编代码；
  - 删减了业务不会用到的寄存器，只保存ABI约定的caller save寄存器；
- 协程源语
  - co_create 创建协程运行环境
  - co_resume 切换协程
  - co_yield 让出协程
- Socket族函数Hook
  - 只对业务声明为非阻塞的fd生效；
  - 非Socket族函数，pipe/eventfd等，可以通过poll接口触发协程切换；
- Libco的协程执行环境
  - 协程**只在本线程内**被执行，不会被迁移到其它线程；
    - 切换协程执行的线程，依赖线程私有变量的执行有可能出错；

Libco改造已有框架的过程

1. **修改RPC框架**，支持协程调度；
2. Review代码**全局变量**、**线程私有变量**的使用，看情况是否需要改造成协程私有变量；
3. Review代码**线程锁**的使用，对于带着锁进行网络调用的行为，需要改造成协程安全的锁；
4. Review程序中**栈的使用**，确保不会有过大栈空间的代码写法；
5. 灰度验证改造；

内部框架的改造

- **改造RPC框架，Worker线程池内加入协程池，业务实体运行在协程上；**
  - 系统启动的时候进程内的协程池大小；
  - 数十倍提升服务并发能力；



同步改造的其它组件支持

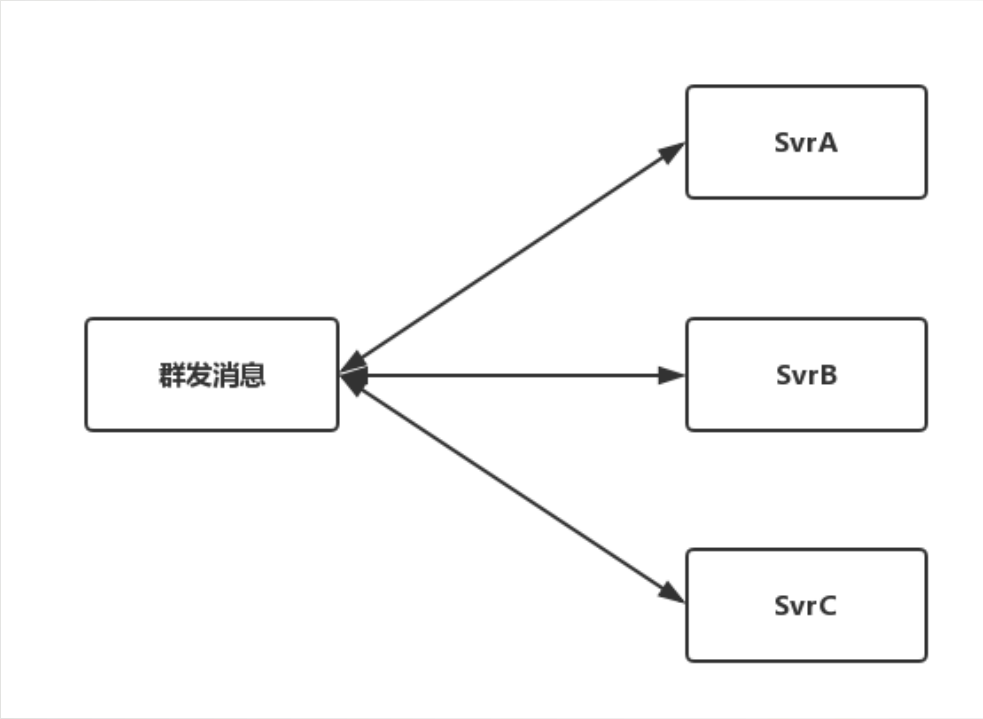
- 协程私有变量
  - 自动适配线程/协程等各种组合环境；
  - 对应线程私有变量，在改造的时候把线程私有变量改造成协程私有变量；

- 跨线程安全的协程锁
  - 自动适配线程/协程等各种组合环境；
  - 通过封装eventfd/pipefd的方式兼容线程/协程的环境；

- 文件的异步实现：
  - 异步文件线程池方式；
    - 工作协程把任务托管给异步线程池，异步线程池负责读写磁盘；

  - DirectIO的方式在本线程完成异步文件读写；
    - 因DirectIO要求页对其，所以使用DirectIO的方式会多读或多写数据；
    - DirectIO无FileCache，需要业务实现缓存与预读策略；

需求：业务需要同时往3台svr发起请求；



- **BatchTask：基于协程池的并发任务执行框架**
  - 使用上，业务以同步的方式编写并发任务(Add + Run)；
  - 任务执行在本线程内的协程池中；
  - 业务一次过压入所有任务，框架通过配置控制同一次Batch的最大并发数，任务列表中的任务会被依次执行；

```
int FuncA(Msg* A)
{
	GetMsgA(A);  //RPC调用
}
int FuncB(Msg* B)
{
	RPCGetMsgB(B); //RPC调用
}
void CallBatch()
{
	Msg a;
	Msg b;
	BatchTaskAddFunc(FuncA, &a); //添加任务A,获取msg a;
	BatchTaskAddFunc(FuncB, *b); //添加任务B,获取msg b;
	BatchTaskRun();  //调用本线程的BatchTask协程池并行执行任务
}
```

实验中的Libco特性

- 提供事件(CoEvent) 回调接口，在协程框架内也可以实现异步网络调用；可以实现协程与异步框架同时执行的效果；
- 改造网络库，把epoll的水平触发改造成边缘触发；
  - 通过中间状态类托管网络时间，减少epoll_ctl的调用次数；
  - 支持协程对fd进行双工通信；
- 实现类Go的Channel类，跨线程/协程通信更便捷；

协程线上运营经验

- **协程栈**大小有限，接入协程的服务谨慎使用栈空间；
- **池化使用协程**，对系统中资源使用心中有数。随手创建与释放协程不是一个好的方式，有可能系统被过多的协程拖垮；
- **协程不适合运行cpu密集型任务**。对于计算较重的服务，需要分离计算线程与网络线程，避免互相影响；
- **过载保护**。对于基于事件循环的协程调度框架，建议监控完成一次事件循环的时间，若此时间过长，会导致其它协程被延迟调度，需要与上层框架配合，减少新任务的调度；

Q & A