# 茜色微服务架构之自实现cRPC

## 服务器框架组成

服务器的由多个模块组成，每个模块负责不同的功能。模块主要分为两类：一类是服务器框架的基础模块，比如日志模块、数据库模块等；另一类是业务逻辑模块，比如兑换模块、关卡模块等。最后这些模块都被编译程动态库，根据配置加载进服务器提供功能。如果其他模块需要使用其他模块，直接将对应模块加载进服务器，进行使用即可。

## 遇到的问题

随着新功能的不断增加以及对原有功能的多次调整，游戏内各个功能需要操作的数据越来越多，使用到的模块也相应增加，模块之间的耦合也越来越高，导致对模块进行改动的维护也越来越难。比如在游戏结束时的战斗结算功能，分别需要经过战斗模块和关卡模块处理，而这两个模块同时又使用了其他很多的模块，这些模块中又分别操作与玩家相关的不同数据：

1. 属性数据：通关消耗和战斗结算获得体力、金钱等；
2. 队伍数据：获取出战队伍卡牌ID的列表等；
3. 卡牌数据：增加出战卡片的技能经验以及通关给与新的卡牌等；
4. 道具数据：通关给与道具奖励等；
5. 活动数据：累计参加活动的次数、累计活动积分、给与活动积分奖励等；
6. 公会数据：活动积分会累加到公会上、给与公会活动积分奖励等；
7. 关卡数据：通关记录以及通关奖励给与等。

原有的逻辑功能的代码是直接操作数据库，将数据从数据库取出、反序列化、修改、序列化后再存入数据库。在项目功能完善的后期，这种方式不仅效率低下，需要频繁的访问数据库，而且不同服务器之间操作数据时的同步也很麻烦，需要使用分布式锁。

## 如何重构

针对上述出现的模块之间耦合过高，数据同步复杂的问题，我们决定对服务器进行重构，采取的方法将数据库进行垂直拆分，将不同业务的数据拆分到不同的数据库，同时保证，指定的业务数据只能由指定的服务器来操作。比如：玩家的公会数据就只能由公会服务器来操作。而原有代码中，需要访问其他业务数据的代码则改用RPC来完成对数据的增删改查。这样就能大大的降低了各个功能模块之间的依赖了，同时避免了使用低效的分布式锁来保证数据同步。

## RPC库的选择

既然确定了重构的方向，那么就可以开始实施重构了。RPC实现库的选择上，我们主要的考虑有两个方面：

1. 支持C++并且支持在协程中使用。我们服务器使用[libgo](https://github.com/yyzybb537/libgo/releases/tag/v2.6)(版本2.6)协程库来提升服务器的承载。所有的消息都会在协程中处理，所以，处理消息时需要使用到的库也必须支持协程；
2. 代码库要成熟稳定的，能够商用的。这个在服务器的稳定性和吞吐量上才能有比较可靠的保障。

根据上面的需求，再结合目前支持C++的RPC的框架库，我们选择了目前比较成熟的gRPC框架，同时使用protobuf作为消息的序列化库。gRPC的版本选的是1.14.1，protobuf的版本选择是3.5.1。

在服务器接入完gRPC后，我们首先在Windows下，进行了基本的功能测试，但是发现两个明显的问题：

1. gRPC的同步调用会阻塞协程所在的线程，直到RPC调用超时或者结果返回时，才取消阻塞；
2. 当RPC结果返回时，程序无法读取返回的结果。

对于上面的第一个问题，我们深入查看了gRPC的代码发现，gRPC的同步调用，底层实现并不是同步的。同步调用时，gRPC实际上是将RPC的调用请求交给gRPC的工作线程，让工作线程去和远程RPC服务器通信。而调用线程和gRPC的工作线程之间是靠锁来同步的，所以，如果只有在有结果时，调用线程才会取消阻塞继续执行。

然而对于上面出现第二个问题，却比第一个问题麻烦得多。我们检查了功能测试的代码，RPC的使用方法和官方的例子相同，但是RPC的结果却无法读取, 而本地抓取网络消息包发现RPC服务器的响应是回复给了调用方的。我们将功能测试的代码从服务器的框架中提取出来单独执行，发现结果又正常了。通过这样的对比我们确定是服务器框架中的某些代码库导致gRPC无法正常工作。最后通过多次对比测试发现，虽然功能测试代码是直接在线程中而不是协程中执行的，但是当服务器加载了协程库后，RPC的调用结果就能够正确读取到了。这个在我们细读了协程库的源代码后确定了这一问题的所在。

我们服务器使用的协程库libgo在程序启动时会hook与网络IO和文件IO相关的API。

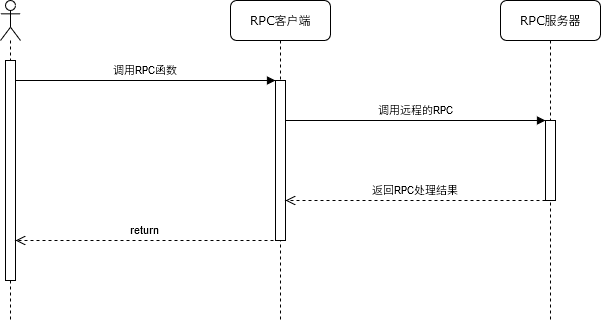
|  |
| --- |
| void coroutine\_hook\_init()  {  ...  // recv-like functions  ok &= XHookAttach((PVOID\*)&recv\_f, &hook\_recv) == NO\_ERROR;  ok &= XHookAttach((PVOID\*)&recvfrom\_f, &hook\_recvfrom) == NO\_ERROR;  ok &= XHookAttach((PVOID\*)&WSARecv\_f, &hook\_WSARecv) == NO\_ERROR;  ok &= XHookAttach((PVOID\*)&WSARecvFrom\_f, &hook\_WSARecvFrom) == NO\_ERROR;  ...  } |

所以，当协程库加载后，调用网络和文件相关的API时，会先调用协程库中hook的函数（所以即使跟踪调试到Windows系统的底层网络API时也无法发现问题)。而协程库在Windows下在判断一个网络套接字是否是阻塞时有BUG，导致无法完全正确的处理Windows下网络IO的接口，从而导致了gRPC的也无法读取到RPC响应结果。这一BUG也在与协程库作者联系后得到了确认。

基于上面使用gRPC库遇到的问题，我们意识到现成的RPC库大多难以在项目中使用，并且现成的支持C++开发的RPC库的选择并不多。最后，结合项目实际的使用情况，我们决定自己实现RPC库。

## 实现自己RPC库……cRPC

对于实现cRPC来说，比较重要的主要有两点，分别是网络库和序列化库。目前项目中使用的网络库是公司技术储备库clib，它提供的网络功能能够很好的在协程中工作，所以cRPC网络库我们决定使用公司技术储备库clib。而序列库方便，我们决定使用protobuf，因为其在易用性和效率上都有不错的表现。



**图4-1 RPC调用的时序图**

在cRPC的实现上，最基本的就是应用程序调用RPC函数，RPC函数内部将调用的参数序列化，通过RPC客户端发送给远程的RPC服务器，并阻塞RPC调用函数直到远程的RPC将结果返回或直至RPC调用超时。因为RPC的调用和处理都在协程中进行的，所以在实现cRPC时，除了要注意一般的RPC的超时异常等问题的处理，还要注意RPC调用和处理都是在协程中进行的。整个cRPC实现过程中主要需要注意的有下面几点：

1. 阻塞RPC的调用方时，只能阻塞调用的协程，不能阻塞调用协程所在的线程，否则将会导致其他协程的无线程执行。在这一点上，我们最初尝试了协程库中提供的Channel用于阻塞同步结果。但是后来压力测试时发现，当每秒RPC调用上千时，Channel的超时读取会失败，即使超时了也不会返回结果继续执行协程后面的代码。所有后来改用了原子变量加协程睡眠的方式模拟限时锁的方式来避免了这个问题。

|  |
| --- |
| bool CoSpinLock::TryLock(const char\* file, int line)  {  if (!lock\_.test\_and\_set(std::memory\_order\_acquire))  {  locked\_ = true;  file\_ = file;  line\_ = line;  return true;  }  return false;  }  bool CoSpinLock::TimedLock(int ms, const char\* file, int line)  {  const auto end = NFTime::GetNowTimeMille()+ ms;  for (;;)  {  if (TryLock(file, line))  {  return true;  }  co\_sleep(3);  const auto now = NFTime::GetNowTimeMille();  if (now >= end)  {  break;  }  }  return false;  } |

**图4-2 RPC同步锁的实现细节**

1. 关于RPC请求的超时处理。因为协程也会有调度。所以，可能RPC请求到达RPC服务器时，就已经超时了。这时可以不用处理RPC请求也直接返回错误。另一个就是RPC处理结果到达RPC客户端时，处理此RPC结果的是与调用RPC不同的另一个协程，而调用RPC的协程可能还在等待，也可能已经超时返回了。两种情况，需要正确的处理或丢弃RPC的结果，并释放之前用于同步RPC结果的锁。
2. 关于RPC失败时的处理方式。RPC失败时处理的方式一般有两种：重试和放弃。重试就是再次调用RPC，直到调用成功或者失败次数达到限制。放弃就是当RPC调用失败时，就像调用方返回错误，而不进行任何重试。因为RPC调用失败时，不确定自动再次调用同样的RPC会不会破坏原本正确的数据，所以我们采取了保守的策略，即放弃策略。

实现完成后的cRPC由客户端模块和服务端模块组成，服务器中的其他模块可以方便的集成使用这些模块。cRPC不仅提供了RPC必备的调用超时控制，而且能完美的和服务器中的协程库协同工作，保证服务器的性能。同时我们也对cRPC进行了细致性能测试和分析。

## 性能测试

在实现了cRPC后我们对其进行了性能测试。测试平台的硬件以及测试的参数如下：

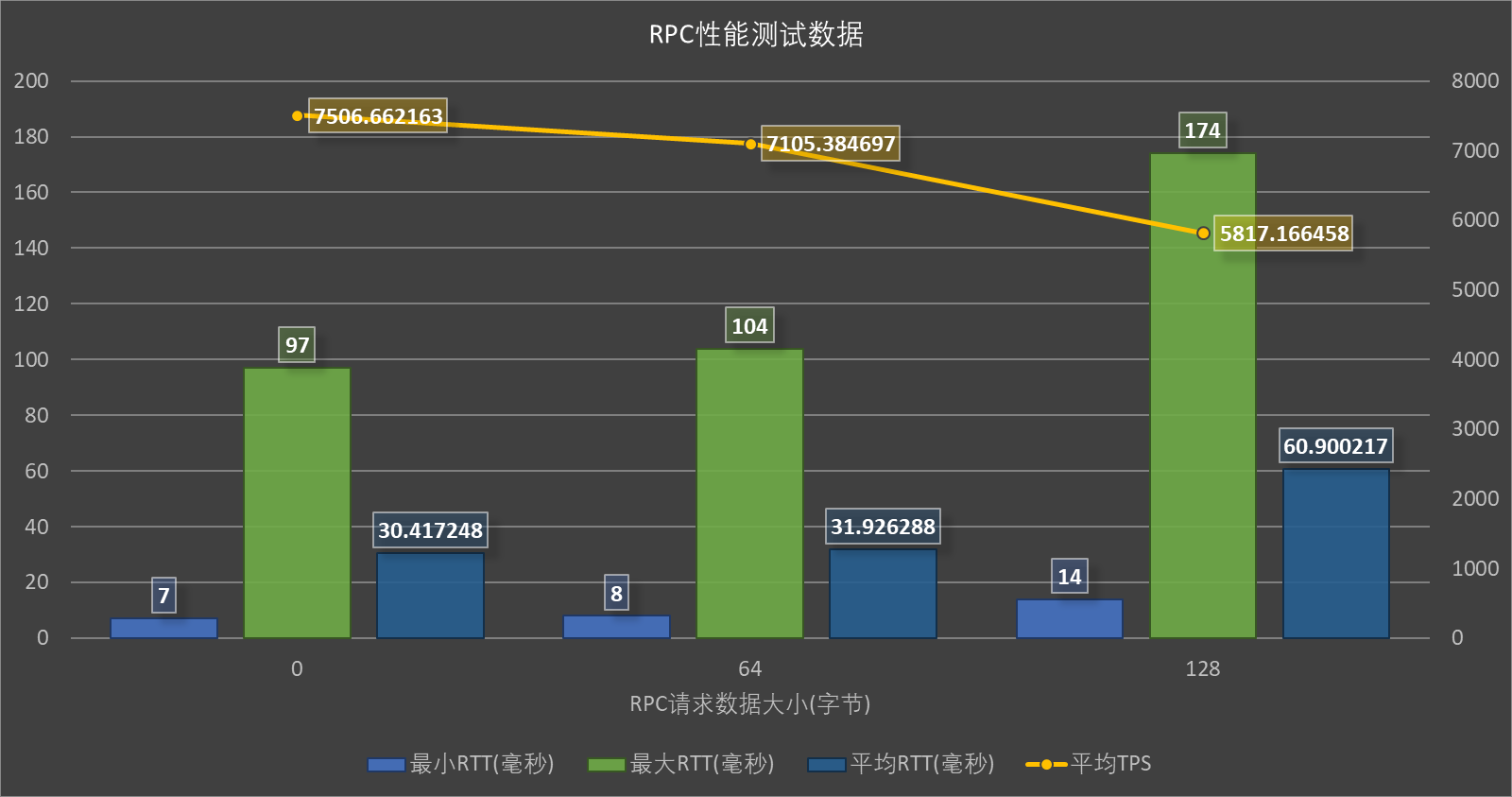
|  |  |
| --- | --- |
| 系统 | Windows 10 Pro |
| CPU | I7-7700@3.6GHz |
| 内存 | 16GB |
| RPC服务器和客户端之间套接字的数量 | 1 |
| RPC服务器和客户端套接字发送缓存大小 | 256KB |
| RPC服务器和客户端套接字发送缓存大小 | 256KB |
| RPC服务器和客户端协程工作线程的数量 | 4 |

**图5-1 RPC性能测试的平台及测试的部分参数**

测试方法如下：

RPC服务器和RPC客户端之间只有一个网络套接字连接，并且他们都在同一台物理机上。性能测试程序每秒调用RPC客户端向RPC服务器发送20000个RPC请求，总共持续30秒。每秒内将20000个RPC请求分100批次发送。并且设置RPC请求的最大处理时间（即RPC服务器开始处理RPC请求的时间与RPC请求创建时间之间差的最大值）为60秒，RPC请求响应的最大时间（即RPC客户端发送RPC请求到收到回复的时间差的最大值）为120秒。RPC服务器收到RPC请求时，将数据原封不动的返回给RPC客户端。分别测试了RPC请求时附带的数据大小分别为0字节、64字节和128字节时的数据。下面是某一次测试的结果：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 请求数据大小（字节） | 平均每秒处理RPC数量 | 最小响应时间（毫秒） | 最大响应时间（毫秒） | 平均响应时间（毫秒） | 响应时间的标准差 |
| 0 | 7506.662163 | 7 | 97 | 30.417248 | 7.661565 |
| 64 | 7105.384697 | 8 | 104 | 31.926288 | 8.638297 |
| 128 | 5817.166458 | 14 | 174 | 60.900217 | 17.224978 |



**图5-2 一次RPC性能测试的结果**

根据性能测试数据，单套接字连接情况，cRPC的TPS基本能够到达7000以上。各个请求的响应时间变化不大，比较稳定，未出现超时或错误等情况。

## 结论

综合上面的数据及分析，我们自己实现的RPC库cRPC能够满足项目目前对RPC使用的功能需求和性能需求。

附性能测试结果的详细数据。

|  |
| --- |
| ----------------------------------------------------------------------------------------  --- Benchmark cost: 79929 millseconds ---  RPC per seconds: 7506.662163  nMinRTT: 7  nMaxRTT: 97  fMeanRTT: 30.417248  fStandardDeviation: 7.661565  Error statistics:  ----------------------------------------------------------------------------------------  ----------------------------------------------------------------------------------------  RPCBenchMarkTest  Coroutine work thread num: 4  nUserID: 70369563868412  nTotalNum: 600000  nPerSecondNum: 20000  nPerSecondBatNum: 100  nHandleTimeoutMS: 60000  nMaxTimeoutMS: 120000  DataSize: 64  ----------------------------------------------------------------------------------------  --- Benchmark cost: 84443 millseconds ---  RPC per seconds: 7105.384697  nMinRTT: 8  nMaxRTT: 104  fMeanRTT: 31.926288  fStandardDeviation: 8.638297  Error statistics:  ----------------------------------------------------------------------------------------  ----------------------------------------------------------------------------------------  RPCBenchMarkTest  Coroutine work thread num: 4  nUserID: 70369563868412  nTotalNum: 600000  nPerSecondNum: 20000  nPerSecondBatNum: 100  nHandleTimeoutMS: 60000  nMaxTimeoutMS: 120000  DataSize: 128  ----------------------------------------------------------------------------------------  --- Benchmark cost: 103143 millseconds ---  RPC per seconds: 5817.166458  nMinRTT: 14  nMaxRTT: 174  fMeanRTT: 60.900217  fStandardDeviation: 17.224978  Error statistics:  ---------------------------------------------------------------------------------------- |