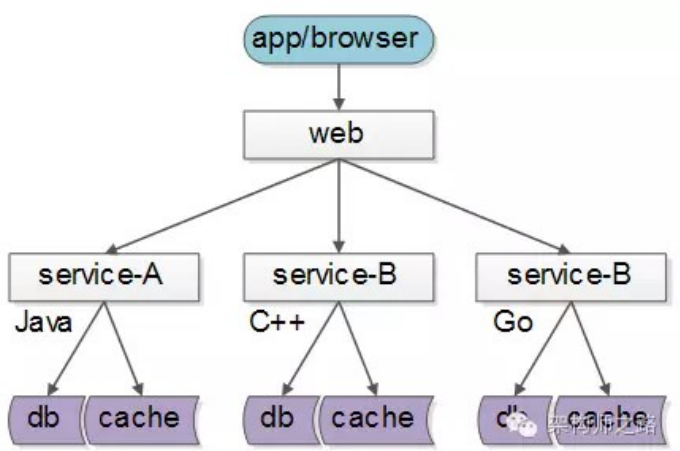
离不开的微服务架构，脱不开的RPC细节（值得收藏）！！！

58沈剑 数据和云 3月14日

微服务离不开RPC框架，RPC框架的原理、实践及细节，是本篇要分享的内容。

服务化有什么好处?

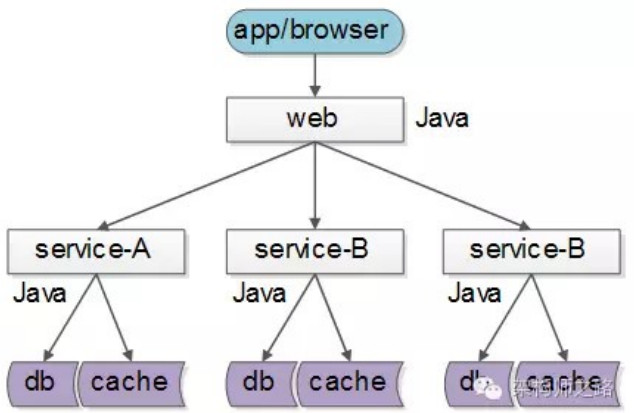
服务化的一个好处就是，不限定服务的提供方使用什么技术选型，能够实现大公司跨团队的技术解耦，如下图所示：



- 服务A：欧洲团队维护，技术背景是Java
- 服务B：美洲团队维护，用C++实现
- 服务C：中国团队维护，技术栈是go

服务的上游调用方，按照接口、协议即可完成对远端服务的调用。

但实际上，大部分互联网公司，研发团队规模有限，大都使用同一套技术体系来实现服务：



这样的话，如果没有统一的服务框架，各个团队的服务提供方就需要各自实现一套**序列化、反序列化、网络框架、连接池、收发线程、超时处理、状态机**等“业务之外”的重复技术劳动，造成整体的低效。

因此，统一服务框架把上述“业务之外”的工作统一实现，是服务化首要解决的问题。

什么是RPC?

Remote Procedure Call Protocol，远程过程调用。

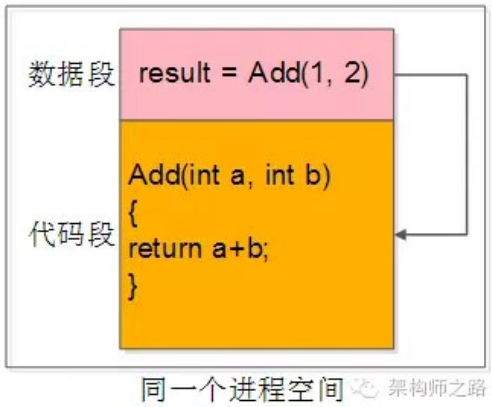
什么是“远程”，为什么“远”?

先来看下什么是“近”，即“本地函数调用”。

当我们写下：

```
int result = Add(1, 2);
```

这行代码的时候，到底发生了什么?

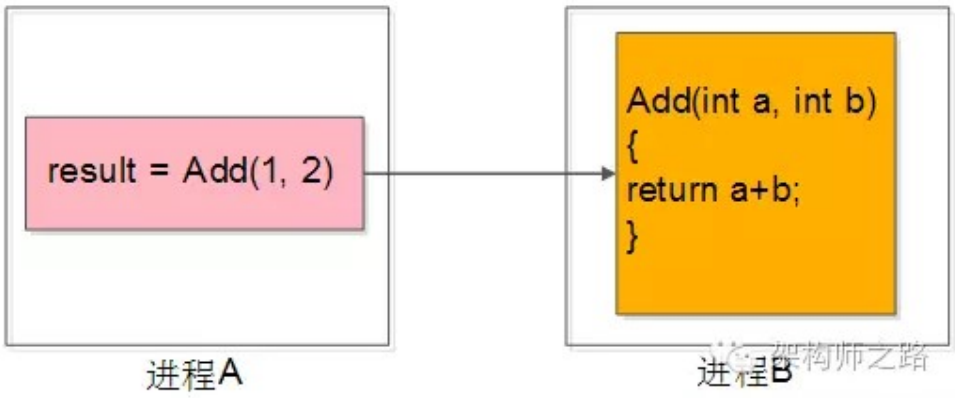


- 传递两个入参
- 调用了本地代码段中的函数，执行运算逻辑
- 返回一个出参

这三个动作，都发生在同一个进程空间里，这是**本地函数调用。**

那有没有办法，调用一个跨进程的函数呢？

典型的，这个进程部署在另一台服务器上。



最容易想到的，两个进程约定一个协议格式，使用Socket通信，来传输：

- 入参
- 调用哪个函数
- 出参

如果能够实现，那这就是“远程”过程调用。

Socket通信只能传递连续的字节流，如何将入参、函数都放到连续的字节流里呢？

假设，设计一个11字节的请求报文：

- 前3个字节填入函数名“add”
- 中间4个字节填入第一个参数“1”
- 末尾4个字节填入第二个参数“2”

同理，可以设计一个4字节响应报文：

- 4个字节填入处理结果“3”

调用方的代码可能变为：

```
request = MakePacket(“add”, 1, 2);
```

```
SendRequest_ToService_B(request);

response = RecieveRespnse_FromService_B();

int result = unMakePacket(respnse);
```

这4个步骤是：

（1）将传入参数变为字节流；

（2）将字节流发给服务B；

（3）从服务B接受返回字节流；

（4）将返回字节流变为传出参数；

服务方的代码可能变为：

```
request = RecieveRequest();

args/function = unMakePacket(request);

result = Add(1, 2);

response = MakePacket(result);

SendResponse(response);
```

这个5个步骤也很好理解：

（1）服务端收到字节流；

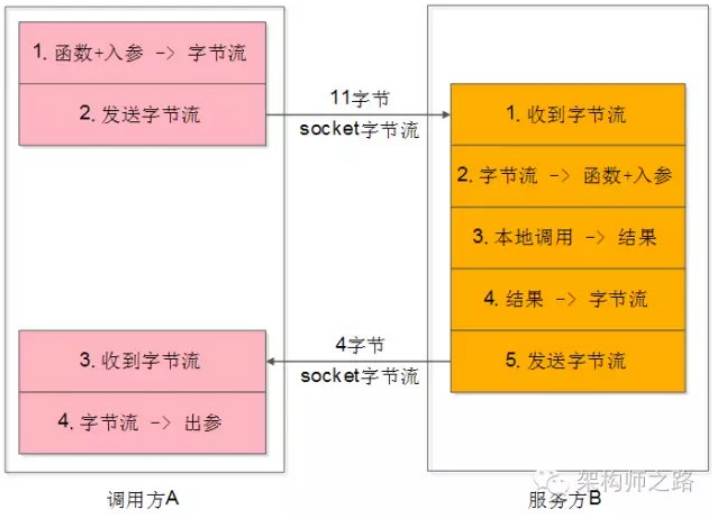
（2）将字节流转为函数名与参数；

（3）本地调用函数得到结果；

（4）将结果转变为字节流；

（5）将字节流发送给调用方；

这个过程用一张图描述如下：



调用方与服务方的处理步骤都是非常清晰。

这个过程存在最大的问题是什么呢？

调用方太麻烦了，每次都要关注很多底层细节：

- 入参到字节流的转化，即序列化应用层协议细节
- socket发送，即网络传输协议细节
- socket接收
- 字节流到出参的转化，即反序列化应用层协议细节

能不能调用层不关注这个细节？

可以，RPC框架就是解决这个问题的，它能够让调用方“像调用本地函数一样调用远端的函数（服务）”。

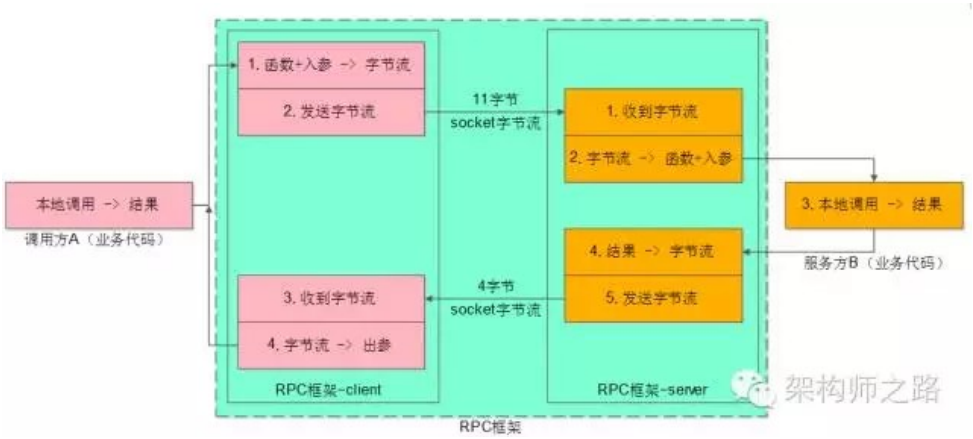
讲到这里，是不是对RPC，对序列化范序列化有点感觉了？往下看，有更多的底层细节。

RPC框架的职责是什么？

RPC框架，要向调用方屏蔽各种复杂性，要向服务提供方也屏蔽各类复杂性：

- 服务调用方client感觉就像调用本地函数一样，来调用服务
- 服务提供方server感觉就像实现一个本地函数一样，来实现服务

所以整个RPC框架又分为**client部分**与**server部分**，实现上面的目标，把复杂性屏蔽，就是RPC框架的职责。



如上图所示，**业务方的职责**是：

- 调用方A，传入参数，执行调用，拿到结果
- 服务方B，收到参数，执行逻辑，返回结果

RPC框架的职责是，中间大蓝框的部分：

- **client端**：序列化、反序列化、连接池管理、负载均衡、故障转移、队列管理，超时管理、异步管理等等
- **server端**：服务端组件、服务端收发包队列、io线程、工作线程、序列化反序列化等

server端的技术大家了解的比较多，接下来重点讲讲client端的技术细节。

先来看看RPC-client部分的“序列化反序列化”部分。

为什么要进行序列化？

工程师通常使用“对象”来进行数据的操纵：

```
class User{
        std::String user_name;
        uint64_t user_id;
        uint32_t user_age;
};

User u = new User(“shenjian”);
u.setUid(123);
u.setAge(35);
```

但当需要对数据进行**存储**或者**传输**时，“对象”就不这么好用了，往往需要把数据转化成连续空间的“二进制字节流”，一些典型的场景是：

- 数据库**索引的磁盘存储**：数据库的索引在内存里是b+树，但这个格式是不能够直接存储到磁盘上的，所以需要把b+树转化为连续空间的二进制字节流，才能存储到磁盘上
- **缓存的KV存储**：redis/memcache是KV类型的缓存，缓存存储的value必须是连续空间的二进制字节流，而不能够是User对象
- **数据的网络传输**：socket发送的数据必须是连续空间的二进制字节流，也不能是对象

所谓**序列化**（Serialization），就是将“对象”形态的数据转化为“连续空间二进制字节流”形态数据的过程。这个过程的逆过程叫做**反序列化**。

怎么进行序列化？

这是一个非常细节的问题，要是让你来把“对象”转化为字节流，你会怎么做？很容易想到的一个方法是xml（或者json）这类具有自描述特性的标记性语言：

```
<class name=”User”>
<element name=”user_name” type=”std::String” value=”shenjian” />
<element name=”user_id” type=”uint64_t” value=”123” />
<element name=”user_age” type=”uint32_t” value=”35” />
</class>
```

规定好转换规则，发送方很容易把User类的一个对象序列化为xml，服务方收到xml二进制流之后，也很容易将其范序列化为User对象。

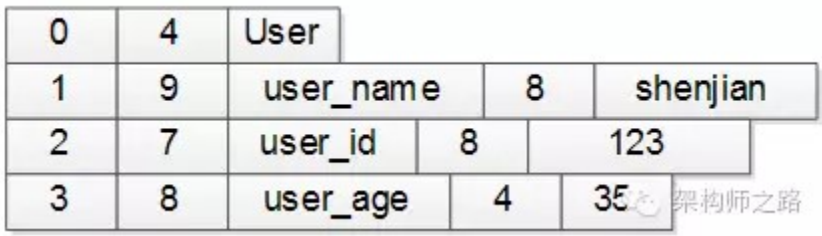
画外音：语言支持反射时，这个工作很容易。

第二个方法是自己实现二进制协议来进行序列化，还是以上面的User对象为例，可以设计一个这样的通用协议：

- 头4个字节表示序号
- 序号后面的4个字节表示key的长度m
- 接下来的m个字节表示key的值

- 接下来的4个字节表示value的长度n
- 接下来的n个字节表示value的值
- 像xml一样递归下去，直到描述完整个对象

上面的User对象，用这个协议描述出来可能是这样的：



- 第一行：序号4个字节（设0表示类名），类名长度4个字节（长度为4），接下来4个字节是类名（“User”），共12字节
- 第二行：序号4个字节（1表示第一个属性），属性长度4个字节（长度为9），接下来9个字节是属性名（“user_name”），属性值长度4个字节（长度为8），属性值8个字节（值为“shenjian”），共29字节
- 第三行：序号4个字节（2表示第二个属性），属性长度4个字节（长度为7），接下来7个字节是属性名（“user_id”），属性值长度4个字节（长度为8），属性值8个字节（值为123），共27字节
- 第四行：序号4个字节（3表示第三个属性），属性长度4个字节（长度为8），接下来8个字节是属性名（“user_name”），属性值长度4个字节（长度为4），属性值4个字节（值为35），共24字节

整个二进制字节流共12+29+27+24=92字节。

实际的序列化协议要考虑的细节远比这个多，例如：强类型的语言不仅要还原属性名，属性值，还要还原属性类型；复杂的对象不仅要考虑普通类型，还要考虑对象嵌套类型等。无论如何，序列化的思路都是类似的。

序列化协议要考虑什么因素?

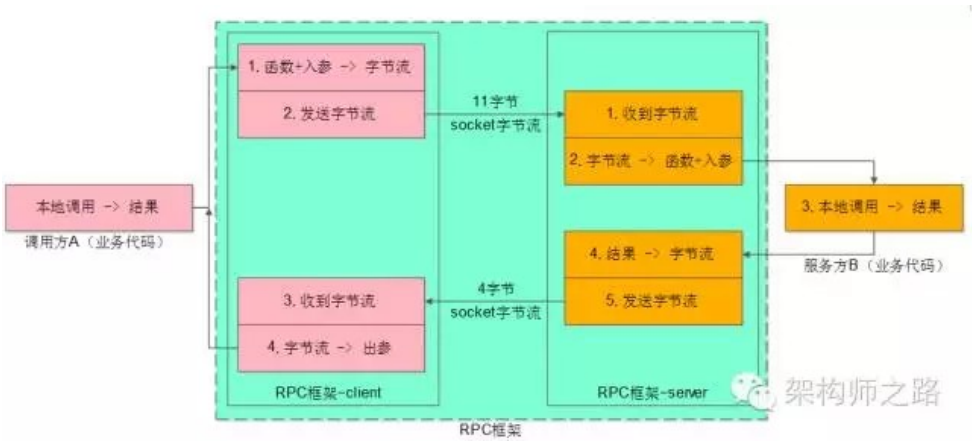
不管使用成熟协议xml/json，还是自定义二进制协议来序列化对象，序列化协议设计时都需要考虑以下这些因素。

- **解析效率**：这个应该是序列化协议应该首要考虑的因素，像xml/json解析起来比较耗时，需要解析doom树，二进制自定义协议解析起来效率就很高
- **压缩率，传输有效性**：同样一个对象，xml/json传输起来有大量的xml标签，信息有效性低，二进制自定义协议占用的空间相对来说就小多了

- **扩展性与兼容性**：是否能够方便的增加字段，增加字段后旧版客户端是否需要强制升级，都是需要考虑的问题，xml/json和上面的二进制协议都能够方便的扩展
- **可读性与可调试性**：这个很好理解，xml/json的可读性就比二进制协议好很多
- **跨语言**：上面的两个协议都是跨语言的，有些序列化协议是与开发语言紧密相关的，例如dubbo的序列化协议就只能支持Java的RPC调用
- **通用性**：xml/json非常通用，都有很好的第三方解析库，各个语言解析起来都十分方便，上面自定义的二进制协议虽然能够跨语言，但每个语言都要写一个简易的协议客户端

有哪些常见的序列化方式？

- xml/json：解析效率，压缩率都较差，扩展性、可读性、通用性较好
- thrift
- protobuf：Google出品，必属精品，各方面都不错，强烈推荐，属于二进制协议，可读性差了点，但也有类似的to-string协议帮助调试问题
- Avro
- CORBA
- mc_pack：懂的同学就懂，不懂的就不懂了，09年用过，传说各方面都超越protobuf，懂行的同学可以说一下现状
- ...



RPC-client除了：

- 序列化反序列化的部分（上图中的1、4）

还包含：

- 发送字节流与接收字节流的部分（上图中的2、3）

这一部分，又分为同步调用与异步调用两种方式，下面一一来进行介绍。

画外音：搞通透RPC-client确实不容易。

同步调用的代码片段为：

```
Result = Add(Obj1, Obj2);// 得到Result之前处于阻塞状态
```

异步调用的代码片段为：

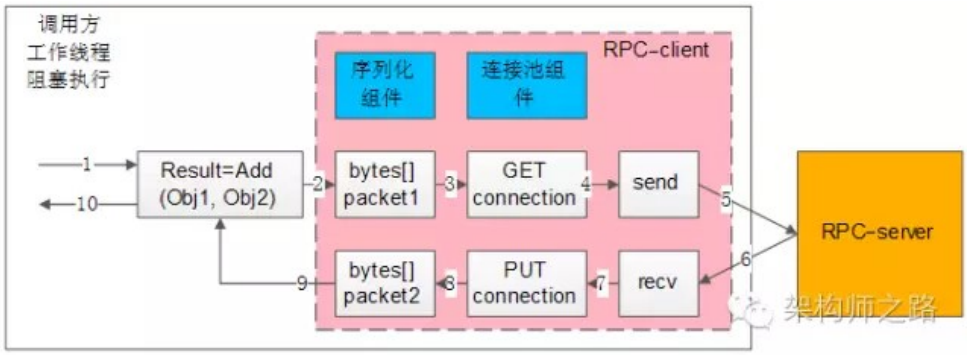
```
Add(Obj1, Obj2, callback);// 调用后直接返回，不等结果
```

处理结果通过回调为：

```
callback(Result){// 得到处理结果后会调用这个回调函数
        …
}
```

这两类调用，在RPC-client里，实现方式完全不一样。

RPC-client同步调用架构如何？



所谓同步调用，在得到结果之前，一直处于阻塞状态，会一直占用一个工作线程，上图简单的说明了一下组件、交互、流程步骤：

- **左边大框**，代表了调用方的一个工作线程
- 左边**粉色中框**，代表了RPC-client组件
- 右边**橙色框**，代表了RPC-server
- **蓝色两个小框**，代表了同步RPC-client两个核心组件，序列化组件与连接池组件
- **白色的流程小框**，以及箭头序号1-10，代表整个工作线程的串行执行步骤：

1）业务代码发起RPC调用：

```
Result=Add(Obj1,Obj2)
```

2）序列化组件，将对象调用序列化成二进制字节流，可理解为一个待发送的包packet1；

3）通过连接池组件拿到一个可用的连接connection；

4）通过连接connection将包packet1发送给RPC-server；

5）发送包在网络传输，发给RPC-server；

6）响应包在网络传输，发回给RPC-client；

7）通过连接connection从RPC-server收取响应包packet2；

8）通过连接池组件，将conneciont放回连接池；

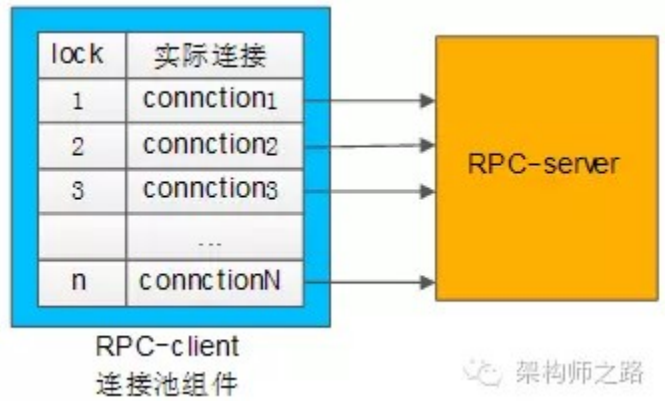
9）序列化组件，将packet2范序列化为Result对象返回给调用方；

10）业务代码获取Result结果，工作线程继续往下走；

画外音：请对照架构图中的1-10步骤阅读。

连接池组件有什么作用？

RPC框架锁支持的负载均衡、故障转移、发送超时等特性，都是通过连接池组件去实现的。



典型连接池组件对外提供的接口为：

```
int ConnectionPool::init(…);

Connection ConnectionPool::getConnection();

int ConnectionPool::putConnection(Connection t);
```

init做了些什么？

和下游RPC-server（一般是一个集群），建立N个tcp长连接，即所谓的连接“池”。

getConnection做了些什么？

从连接“池”中拿一个连接，加锁（置一个标志位），返回给调用方。

putConnection做了些什么？

将一个分配出去的连接放回连接“池”中，解锁（也是置一个标志位）。

如何实现负载均衡？

连接池中建立了与一个RPC-server集群的连接，连接池在返回连接的时候，需要具备随机性。

如何实现故障转移？

连接池中建立了与一个RPC-server集群的连接，当连接池发现某一个机器的连接异常后，需要将这个机器的连接排除掉，返回正常的连接，在机器恢复后，再将连接加回来。

如何实现发送超时？

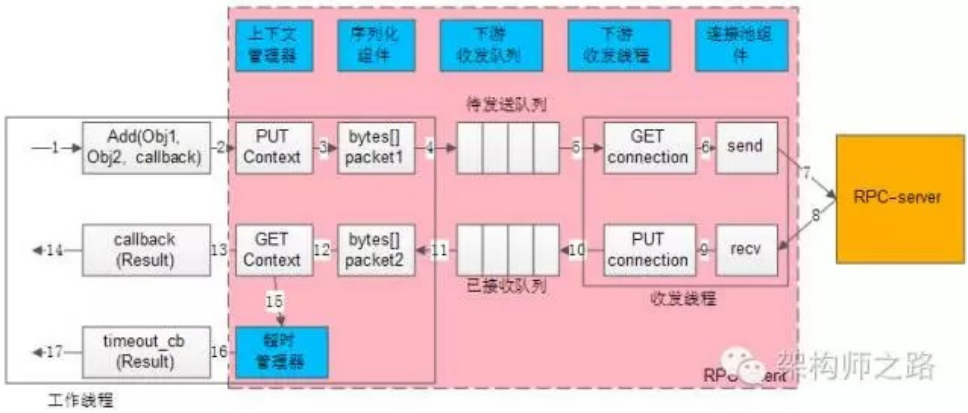
因为是同步阻塞调用，拿到一个连接后，使用带超时的send/recv即可实现带超时的发送和接收。

总的来说，同步的RPC-client的实现是相对比较容易的，序列化组件、连接池组件配合多工作线程数，就能够实现。

遗留问题，**工作线程数设置为多少最合适？**

这个问题在《工作线程数究竟要设置为多少最合适？》中讨论过，此处不再深究。

RPC-client异步回调架构如何？



所谓异步回调，在得到结果之前，不会处于阻塞状态，理论上任何时间都没有任何线程处于阻塞状态，因此异步回调的模型，理论上只需要很少的工作线程与服务连接就能够达到很高的吞吐量，如上图所示：

- **左边的框框**，是少量工作线程（少数几个就行了）进行调用与回调
- **中间粉色的框框**，代表了RPC-client组件
- **右边橙色框**，代表了RPC-server
- **蓝色六个小框**，代表了异步RPC-client六个核心组件：**上下文管理器，超时管理器，序列化组件，下游收发队列，下游收发线程，连接池组件**
- **白色的流程小框**，以及箭头序号1-17，代表整个工作线程的串行执行步骤：

1）业务代码发起异步RPC调用；

```
Add(Obj1,Obj2, callback)
```

2）上下文管理器，将请求，回调，上下文存储起来；

3）序列化组件，将对象调用序列化成二进制字节流，可理解为一个待发送的包packet1；

4）下游收发队列，将报文放入“待发送队列”，此时调用返回，不会阻塞工作线程；

5）下游收发线程，将报文从“待发送队列”中取出，通过连接池组件拿到一个可用的连接connection；

6）通过连接connection将包packet1发送给RPC-server；

7）发送包在网络传输，发给RPC-server；

8）响应包在网络传输，发回给RPC-client；

9）通过连接connection从RPC-server收取响应包packet2；

10）下游收发线程，将报文放入“已接受队列”，通过连接池组件，将conneciont放回连接池；

11）下游收发队列里，报文被取出，此时回调将要开始，不会阻塞工作线程；

12）序列化组件，将packet2范序列化为Result对象；

13）上下文管理器，将结果，回调，上下文取出；

14）通过callback回调业务代码，返回Result结果，工作线程继续往下走；

如果请求长时间不返回，处理流程是：

15）上下文管理器，请求长时间没有返回；

16）超时管理器拿到超时的上下文；

17）通过timeout_cb回调业务代码，工作线程继续往下走；

画外音：请配合架构图仔细看几遍这个流程。

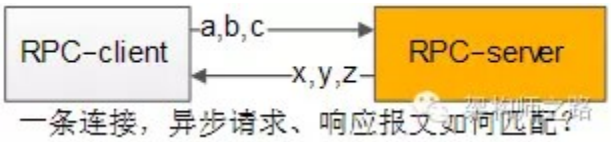
序列化组件和连接池组件上文已经介绍过，收发队列与收发线程比较容易理解。下面重点介绍**上下文管理器**与**超时管理器**这两个总的组件。

为什么需要上下文管理器？

由于请求包的发送，响应包的回调都是异步的，甚至不在同一个工作线程中完成，需要一个组件来记录一个请求的上下文，把请求-响应-回调等一些信息匹配起来。

如何将请求-响应-回调这些信息匹配起来？

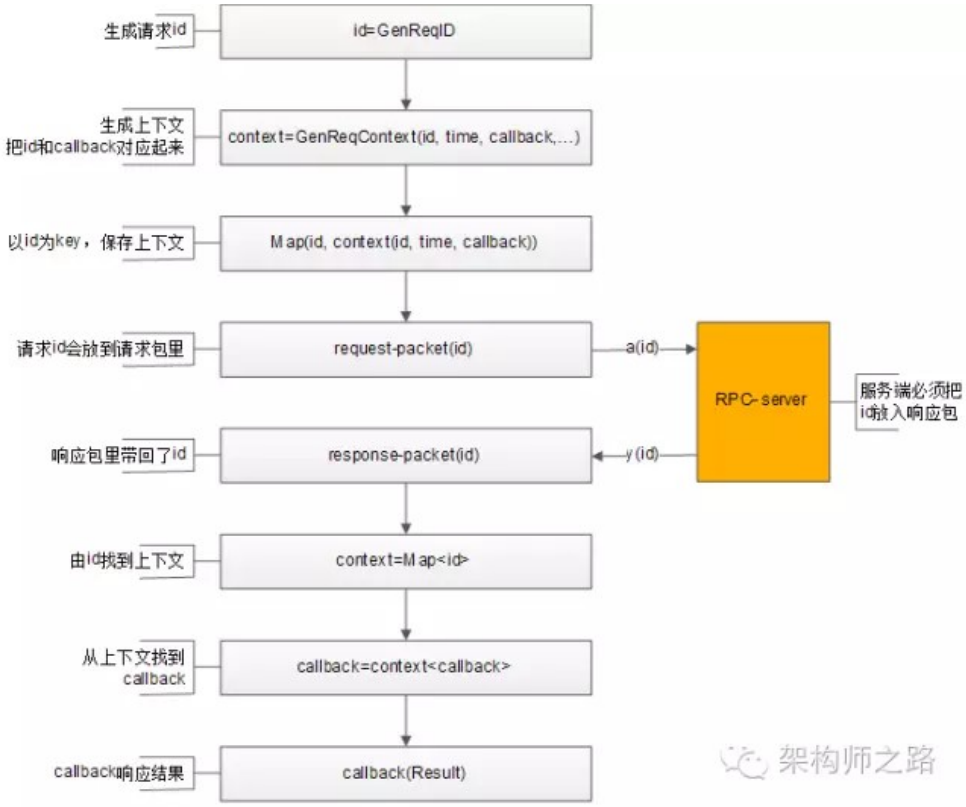
这是一个很有意思的问题，通过一条连接往下游服务发送了a，b，c三个请求包，异步的收到了x，y，z三个响应包：



怎么知道哪个请求包与哪个响应包对应？

怎么知道哪个响应包与哪个回调函数对应？

可以通过“请求id”来实现请求-响应-回调的串联。



整个处理流程如上，通过请求id，上下文管理器来对应请求-响应-callback之间的映射关系：

1）生成请求id；

2）生成请求上下文context，上下文中包含发送时间time，回调函数callback等信息；

3）上下文管理器记录req-id与上下文context的映射关系；

4）将req-id打在请求包里发给RPC-server；

5）RPC-server将req-id打在响应包里返回；

6）由响应包中的req-id，通过上下文管理器找到原来的上下文context；

7）从上下文context中拿到回调函数callback；

8）callback将Result带回，推动业务的进一步执行；

如何实现负载均衡，故障转移？

与同步的连接池思路类似，不同之处在于：

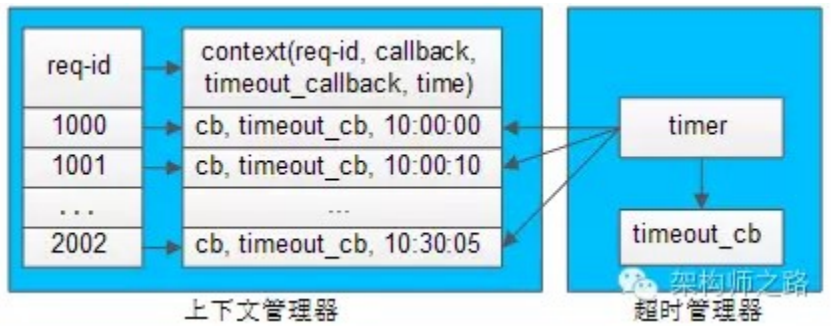
- 同步连接池使用阻塞方式收发，需要与一个服务的一个ip建立多条连接
- 异步收发，一个服务的一个ip只需要建立少量的连接（例如，一条tcp连接）

如何实现超时发送与接收？

超时收发，与同步阻塞收发的实现就不一样了：

- 同步阻塞超时，可以直接使用带超时的send/recv来实现
- 异步非阻塞的nio的网络报文收发，由于连接不会一直等待回包，超时是由超时管理器实现的

超时管理器如何实现超时管理？



超时管理器，用于实现请求回包超时回调处理。

每一个请求发送给下游RPC-server，会在上下文管理器中保存req-id与上下文的信息，上下文中保存了请求很多相关信息，例如req-id，回包回调，超时回调，发送时间等。

超时管理器启动timer对上下文管理器中的context进行扫描，看上下文中请求发送时间是否过长，如果过长，就不再等待回包，直接超时回调，推动业务流程继续往下走，并将上下文删除掉。

如果超时回调执行后，正常的回包又到达，通过req-id在上下文管理器里找不到上下文，就直接将请求丢弃。

画外音：因为已经超时处理了，无法恢复上下文。

无论如何，异步回调和同步回调相比，除了序列化组件和连接池组件，会多出上下文管理器，超时管理器，下游收发队列，下游收发线程等组件，并且对调用方的调用习惯有影响。

画外音：编程习惯，由同步变为了回调。

异步回调能提高系统整体的吞吐量，具体使用哪种方式实现RPC-client，可以结合业务场景来选取。

总结

什么是RPC调用?

像调用本地函数一样，调用一个远端服务。

为什么需要RPC框架?

RPC框架用于屏蔽RPC调用过程中的序列化，网络传输等技术细节。让调用方只专注于调用，服务方只专注于实现调用。

什么是序列化？为什么需要序列化?

把对象转化为连续二进制流的过程，叫做序列化。磁盘存储，缓存存储，网络传输只能操作于二进制流，所以必须序列化。

同步RPC-client的核心组件是什么?

同步RPC-client的核心组件是序列化组件、连接池组件。它通过连接池来实现负载均衡与故障转移，通过阻塞的收发来实现超时处理。

异步RPC-client的核心组件是什么?

异步RPC-client的核心组件是序列化组件、连接池组件、收发队列、收发线程、上下文管理器、超时管理器。它通过“请求id”来关联请求包-响应包-回调函数，用上下文管理器来管理上下文，用超时管理器中的timer触发超时回调，推进业务流程的超时处理。

思路比结论重要。

转载自：架构师之路

DBA04，DBA 手记4 电子书
122ARCH，Oracle 12.2体系结构图
2018OOW，Oracle OpenWorld 资料


zData
Total DB Time
IO 状态
Back Up
Restore
zData Light Storage Client
基于x86架构的Oracle RAC 提供计算服务
Oracle RAC
InfiniBand交换机
提供高速互联
X86 SERVER
zData Light Storage Server
PCIe闪存卡和SSD提供高容量，高性能存储资源
超融合版
zData Fusion
zData Light Compute & Storage Server
精简版
zData Lite
仲裁
节点
zData Light Compute & Storage Server
嘉实基金
Harvest Fund
石嘴山银行
BANK OF SHIZUISHAN
中国移动
China Mobile
数据和云


云和恩墨大讲堂 | 一个分享交流的地方
长按，识别二维码，加关注
请备注：云和恩墨大讲堂
数据和云