Connection 的基本设计

Connection 是 Client 端的核心组件，用于 client 向 server 发出 request，接收 response。Connection 是共享组件，client 同一个特定的 server 间用一个 Connection 维持链接。而多个 Fiber 如果向同一个 server 发出 request，都只能通过同一个 Connection 实例执行通讯。

因而，Connection 必须能够处理多 fiber 共享的能力。由于共享的存在，必须对不同 Fiber 发出的 request 和相对应的 response 做同步处理。对于 request 和 response（统一称为 mail）的同步依赖于 mail header 中的 mailno 实现。mailno 是一个递增的流水号，request 在 server 端得到处理后，response 将携带相应的 mailno 返回。Connection 依赖此 mailno 回溯到发出 request 的 fiber，将其唤醒。

Connection 的核心功能是 send()和 recv()，分别用于发出 request 和接受 response。典型的操作是先通过 send()发出 request，然后调用 recv()等待 response()的返回。Fiber 在 recv()调用后，会被挂起，处于等待状态，直到 response 返回。

Connection::send()

send()所执行的操作比较简单，首先生成 mailno（将++Connection::currMailNo），然后构造 mail，并将其送入一个 DataQueue。Connection 通过这个 DataQueue 将来自各 Fiber 的 mail（即 request）缓存起来，由一独立的 fiber 负责发送，以实现共享[1]。最后，send()返回 mailno。mailno 将被 recv()使用。

Connection::recv()

recv()执行等待 response，并解析 mail。

recv()首先获取 recv buffer 中对应的 slot（具体做法详见后文）。然后锁住该 slot，将当前 fiber，数据缓存地址等填入 slot。最后 yield()。

recv()被唤醒之后，将 slot 的 mailno 设置成 invalid 值，表示 slot 当前空闲。

request/response 匹配

request/response 的匹配实际上就是被挂起的 recv()调用与 response 的匹配。即发出 request 后，调用 recv()进入等待状态（没有 response 的 request，则无须调用 recv()），相应的 fiber 等参数应被保存起来。当 response 到达时，需要能够找到对应的 fiber 等参数，执行数据操作，最后唤醒 fiber。

前文已述，request 通过 mailno 同 response 相关联，response 会携带 mailno 返回。因而，可以通过 mailno 同 fiber 等参数关联。具体方案如下：

建立 recv buffer。recv buffer 是一个固定大小（SlotCount）的 slot 数组。每个 slot 包含 fiber、数据缓存等数据，同时还带有一个 mutex（FiberMutex）。

当 recv()被调用时，mailno 会做为参数传入。recv()将 mailno%SlotCount，得到它在 recv()中的 slot 位置。取出 slot 后，首先将其 lock。这样如果 slot 已被占用，后来的操作将会被排队。之后，recv()便可向 slot 填入 fiber 等参数。完成后，通过 yield()将自己挂起。

读取 response 的是一个独立的 receive fiber，从 socket 中循环提取 response 信息。取得一个 response 后，从中提取出 mailno，通过%SlotCount 得到 slot。然后 lock mailno，而后检测 mailno 是否有效。如果有效，就执行反序列化。否则，抛弃 mail。随后将 mailno unlock。最后，唤醒 fiber。

---

1 使用 FiberMutex 是另一种共享 Connection::send()的手段。但相比之下，使用 DataQueue 性能更好。

数据的 **unserialize**

recv()的一个重要操作是将来自 socket 的数据反序列化。有两个位置可执行反序列化：1、recv()中，yield()之后；2、receive fiber 中。

在 1 中执行，代码简单，但存在 2 个问题：

1、mail 中缺少报文大小的信息，因而 receive fiber 无法知道该提取多大的数据传递给 recv()。

2、在读取反序列化的过程中，可能发生缓冲区读空的情况，因而需要向 socket 读取数据，这将发起异步操作，并将 recv()挂起。如果此时又另一 fiber 执行序列化，它将会读到错误的数据。

问题 2，可以通过锁住 socket 的方式防止误读。但问题 1 却无法绕过，因为 recv()和 receive fiber 无法知道数据反序列化之前的大小，这完全由反序列化函数控制。即使可以得到这个大小，receive fiber 传递出来的数据是反序列化之前的数据，还需要 recv()进一步执行反序列化。这样多了一次数据复制的操作，降低性能。

因此，反序列化必须在 receive fiber 中进行。但 receive fiber 并不知道该如何反序列化。因此，需要 recv()通过 slot 将反序列化的执行函数传递给 receive fiber，同时传递的还有存放反序列化后数据的缓冲区指针。receive fiber 在取得 response，并找到对应 slot 之后，调用 recv()提供的反序列化函数。完成后，才将 fiber 唤醒。

timed_recv()

recv()没有 timeout。如果 server 长时间没有回应，recv()将一直阻塞下去。

为处理这种情况，Connection 另外提供了 timed_recv()。timed_recv()允许设定一个 timeout，在 server 长期没有响应之后，超时退出阻塞。

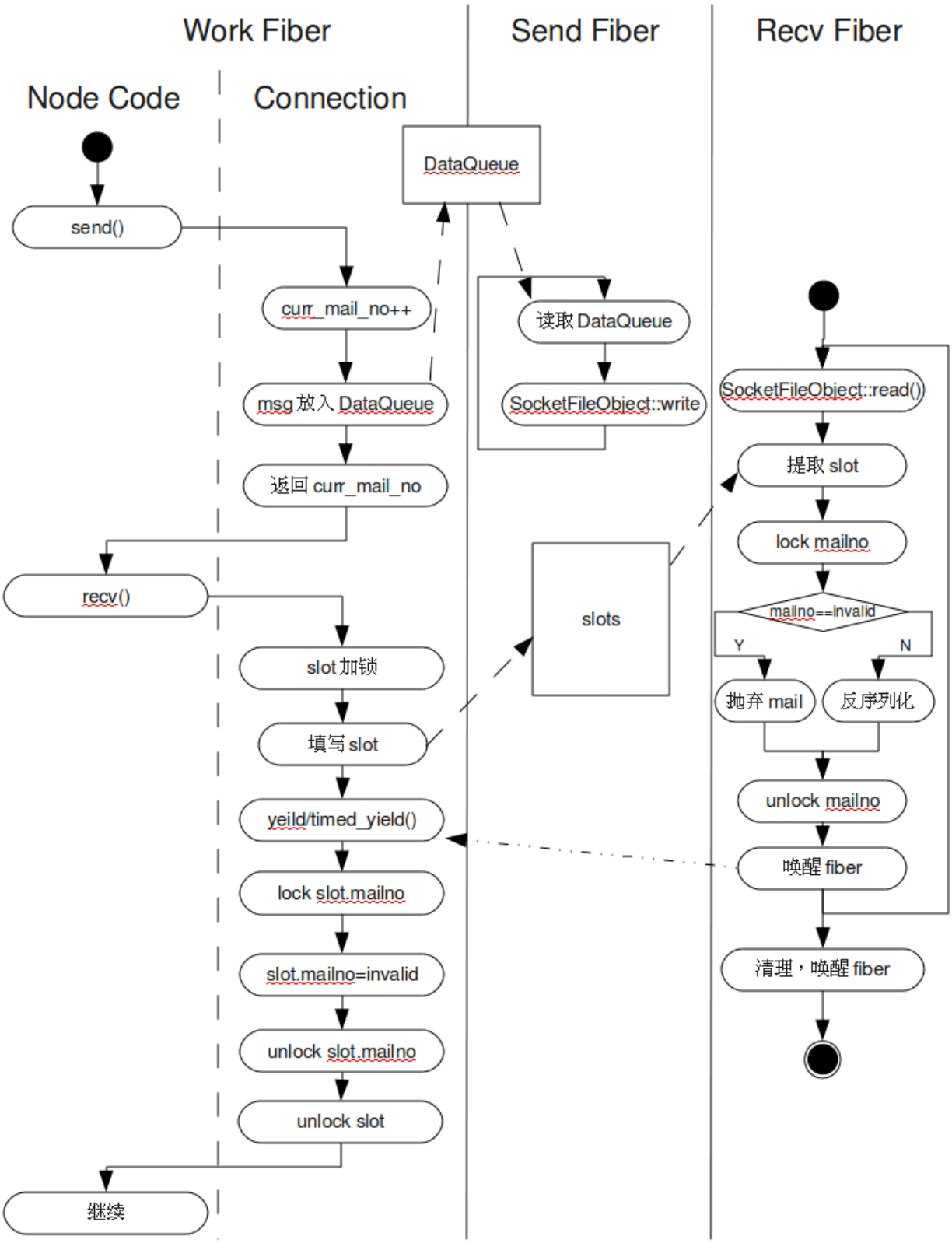
timed_recv()的实现与 recv()类似。差别在于用 timed_yield()代替了 yield()。前者提供了超时唤醒的功能。timed_yield()有 bool 返回值，false 表示 timeout，true 表示正常唤醒。timed_recv()根据不同的返回值做相应的处理。

设置 mailno 的操作与 recv()有所差异，在设置 mailno 之前将其 lock，之后再 unlock。

清理

当 Connection 的 receive fiber 遇到 socket 错误时，便会退出运行。（socket 将被关闭）。在退出运行前，receive fiber 必须执行清理工作，激活 recv buffer 中所有等待的 fiber。

receive fiber 退出循环后，开始执行清理：遍历每一个 slot，将 fiber 唤醒。已唤醒的 fiber 会在释放 mutex 的时候，将 slot 上排队的后续 fiber 也会被唤醒。


Work Fiber
Send Fiber
Recv Fiber
Node Code
Connection
DataQueue
send()
curr_mail_no++
msg 放入 DataQueue
返回 curr_mail_no
recv()
slot 加锁
填写 slot
yeild/timed_yield()
lock slot.mailno
slot.mailno=invalid
unlock slot.mailno
unlock slot
继续
读取 DataQueue
SocketFileObject::write
slots
SocketFileObject::read()
提取 slot
lock mailno
mailno==invalid
Y
N
抛弃 mail
反序列化
unlock mailno
唤醒 fiber
清理，唤醒 fiber