NaiveNet 应用通信组件使用文档

VERSION:Beta2.0
NAIVENET.DOMOE.CN

目录

欢迎你

首先欢迎你阅读本文档，如果需要寻求帮助可以访问我们的官方帮助页面：http://naivenet.domoe.cn/help.html 若本文档存在写作与技术错误，请使用电子邮件反馈给作者，Email：admin@liuxinyumo.cn

什么是 NaiveNet

在了解一个引擎或者框架之前，我们必须要了解它的作用与目的，这样可以更好地了解自己的需求以便准确的使用它。

NaiveNet 它是基于 NIO 网络模型设计完成的网络通信组件，这套组件可以让我们的原生应用、Web、各类小程序/小游戏以长连接（Socket/Web Socket）通信模式开发，面向需求的高度封装使得开发者无需关注长连接的网络通信细节从而高效的开发应用。如果你想做一个实时交互应用比如一个实时的对战游戏，或者是在线的多平台聊天室，甚至你有更好的实时交互创意，那么学会使用 NaiveNet 会让你事半功倍。

NaiveNet 那些能让你眼前一亮的黑科技

接触新事物，在不了解它的时候，内心总是多少会让人抵触，我们并不知道学会使用它到底有哪些好处，本节将简单的说明 NaiveNet 能为我们带来什么？

- NaiveNet 是跨平台的。所以你不需要担心 Server 是装到 Linux、Mac OS 还是 Windows 而烦恼，总之都支持。
- NaiveNet 是跨语言的。采用 TCP/IP 通信协议，因此语言平台都是通用的，目前作者已经适配了 Java 语言、Python 语言以及 C#语言，选择你喜欢的语言进行高效的开发。
- NaiveNet 与生俱来的分布式架构。天生的分布式架构使得开发变得随心所欲，根据业务需要随意的增减配置，NaiveNet 从一开始就对你产品未来的成功做好了准备。
- NaiveNet 热部署。既然是分布式架构，在对他增减配时自然可以做到用户侧无感知的变更。
- 事件驱动的 Reactor 模型。使用 NaiveNet 你根本不需要关注线程的创建、回收、锁问题，它已经帮你做的很好了，尽管去处理业务就好。
- 两端对等的事件请求模型。像使用 Ajax 请求一样使用全双工通信完成事件处理，这让你的开发思路也变得太清晰了吧！
- 断线恢复能力。客户端是不稳定的，但是 NaiveNet 有很好的断网恢复能力，帮助你化繁为简。
- 三合一协议处理。NaiveNet 为初级开发者做了太多友好的设定，HTTP(S)、WS(S)、Naïve 自有协议并行解析。再也不怕两个协议就需要部署两个端口，特别是很多平台只限定 443 端口。

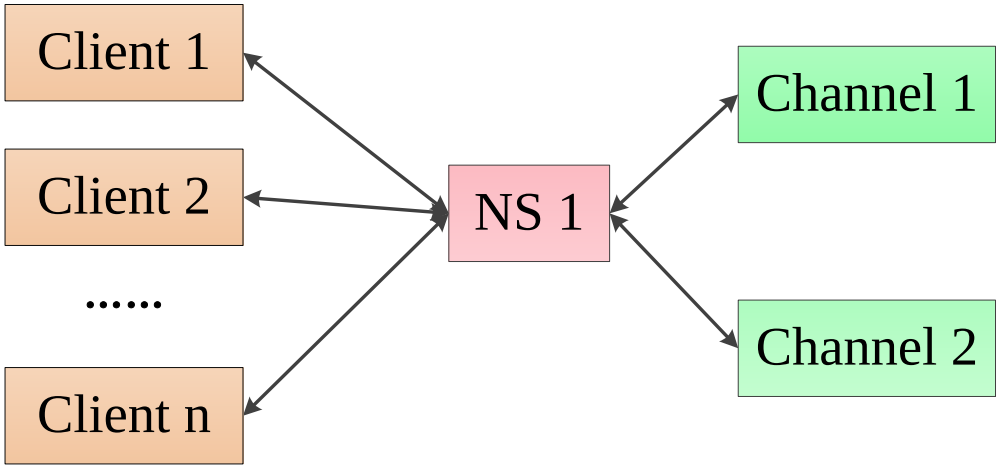
- 业务与服务的多对多能力。你可以将一个业务对应到一台服务器上(1 对 1)，你还可以将多个业务对应的一台服务器上（多对 1），你更可以借助与生俱来的分布式能力将你的业务部署到多台服务器中（多对多），再结合上一个特性 NaiveNet 帮助初级开发者做了很好的成本节约。

新手上路

NaiveNet 的工作原理

了解 NaiveNet 的整体工作原理，会让你更清晰它的使用。

NaiveNet 工作时我们可以理解为 3 个作用方,即:客户端(NaiveNetClient)、网关服务器（NaiveNetServer）与业务服务器（NaiveNetChannel）。很明显我已经为它们起好了名字，在下文中，客户端我们将简称为 Client，网关服务器为 NS，业务服务为 NC。那么一个常见的部署结构可能是这个样子：



从这张图我们就可以很轻松的了解到客户端实际上是与 NS 进行通信，NS 根据需要将请求派发至对应的 NC。NC 与 Client 直接并没有直接通信，这就是为什么说 NaiveNet 组件是与生俱来的分布式结构。

其中对于开发者而言，Client 与 NC 需要引用对应语言平台的 NaiveNet 插件,根据业务需要自行进行开发;而 NS 您只需要简单的几行配置并启动就好了。从上面的部署结构我们可以想到，一个业务的展开流程应该是：先要配置 NS，然后启动它，然后运行 NC（NC 是开发者自行开发的），之后用户的客户端就可以与 NS 建立连接，NS 根据需要派发至 NC，NC 就是开发者自定义的业务处理了。

配置 NaiveNetServer(NS)

NS 并不需要开发者对他进行开发，和 Nginx、Apache 这类网络中间件一样，你只需要对它配置并启动就可以。首先访问 http://naivenet.domoe.cn/ 下载 NaiveNetServer（NS）的文件包，文件包内包含 1 个 Jar 文件与 2 个 JSON 格式

的配置文件。

如果你在跟随着本教程初次使用，两个 JSON 格式配置文件你不需要做任何修改，不过作者仍然建议你用代码编辑器打开他们阅读一下。当你在做实际项目时需要参阅 NS 配置文档 根据实际需要配置它们。这两个 JSON 格式的配置文件需要与 Jar 文件保持在同一个目录内。

正如你所看见的，NS 是 Jar 格式文件，所以 NS 是使用 Java 编写的，启动它时需要确保你的 PC 和服务器安装了 JVM（最低且推荐的版本为 jdk1.8），安装 JVM 过程省略。

现在输入以下命令启动 NS 服务：

```
java -jar 文件路径/NaiveNetServer-xxx.jar
```

橙色文字部分是你的 NS 文件的绝对路径。

运行后通过后，打开浏览器访问：http(s)://你的域名或 IP/status 如果页面输出内容为“NaiveNetServer Service started successfully”代表服务启动正常。

如果你使用的是 Linux 系统命令行模式，或在实际项目中部署 NS，你需要使用 nohup 命令运行 NS。你也可以自行将 NS 注册 Linux 服务来启动或重启它。

nohup 命令：

```
nohup java -jar xxxx/NaiveNetServer-xxx.jar &
```

让客户端与 NS 建立第一次连接

配置并启动 NS 后，我们就可以尝试让客户端与 NS 建立连接，本教程客户端部分采用微信小程序演示，下载 NaiveNetClient 的 JavaScript 版本，JS 版本包内包括了 common 版本与 miniProgram 版本，common 版本是使用在 HTML5 中的 JS 版本，<script src='NaiveNetClient.common.xxxx.js'> 就可以直接使用，而 miniProgram 是各大平台的小程序版本，因此我们需要的是 miniProgram 版本文件。

创建一个小程序/游戏，在脚本中编写：

```
//index.js
import NaiveNet from 'lib/NaiveNetClient-MiniProgame.2.0.0';
const ns = new NaiveNet();
ns.connect({
        ns:'ws://127.0.0.1:4000',
        success:(res)=>{
            console.log('连接成功')
        },
        fail:(res)=>{
            console.log(res);
        }
});
```

如果一切正常的话你的控制台应该会立刻打印出“连接成功”四个字。此时我们为 ns 这个操作句柄添加一个事件，

```
ns.onBreak = function(res){
    console.log('连接断开了')
}
```

重新编译运行，我们会发现在出现“连接成功”数秒后，会弹出“连接断开了”。

不要慌张，这并不是 NaiveNet 的 Bug，而是 NS 有意而为。简单的说，NaiveNet 可能会被很多人下载使用，你当然需要确保只有你的客户端才能被 NS 保持连接，这时我将介绍 NaiveNet 的授权机制概念。

NS 的授权机制

由于你的 NS 是暴露在公网供用户访问的，此时 NS 将接收到各种请求的访问，NS 提供了一种授权机制，当一个连接是初次建立时，该链接没有任何的凭证，此时 NS 将对其保留数秒后主动对其释放。换言之，对于属于你自己的合法客户，必须要让该客户在这数秒内完成授权。NaiveNet 的授权操作只有 NC 能够完成，授权的方式将在后文中介绍。

编写你的第一个 NC

接下来将开始开发你的 NC 部分，首先根据你所熟悉的语言选择对应版本的 NaiveNetChannel 插件，本文以 Java 为例，下载 NaiveNetChannel.jar，将该包导入到你的项目工程里（项目工程请使用 UTF-8 编码）。

在你的认为合适的位置（比如 Main 函数）实例化 NaiveNetChannel 类。

```
nc = new NaiveNetChannel(5000,"abcdefg");
```

NaiveNetChannel 类接受两个参数，第一个参数代表你所希望占用的系统端口号，这个端口号可以是任意的非特殊端口，比如 5000、5001、5002……。第二个参数是一个安全口令，只有持有正确口令的 NS 才能与该 NC 建立连接，实际上细心的开发者已经发现，此处填写的端口号与口令，将与之对应是你的 NS 中 NaiveNetChannel.json 配置文件中的某一项。

此时我们将获得一个 NC 对象，我们的后续操作将基于该对象完成。

实例化后我们并没有启动服务，仍然需要一些基础设置，接下来添加一个监听器，用来获得新用户接入的句柄。

```
nc.setOnNewUserListener(new NaiveNetEvent(){
        @Override
        public void on(User user, byte[] data) {
            System.out.println("新的用户到访");
        }
```

```
});
```

这里，被重写的 on 函数内提供的参数 User user，就是新进用户的操作句柄。

最后我们启动 NC。

```
try {
    nc.launch();
} catch (IOException e) {
    System.out.println("NaiveNetChannel启动失败");
    e.printStackTrace();
}
```

此时我们编译并运行你的 NC 程序。然后回到客户端代码部分，我们在刚刚的建立连接的 success 回调中填写如下代码：

```
console.log('连接成功')
ns.request({
    channel:'my_channel_1',
    controller:'test',
    data:'abc',
    complete:(res)=>{
        console.log(res);
    }
})
```

如果你开发过小程序，是否看到 ns.request({ … })有一种格外的熟悉感？没错，作者也正是出于这样的目的使得你更容易去上手 NaiveNet。此时编译运行你的小程序客户端，如果不出意外的话，你的客户端控制台(Console)将连续打印出三条消息：

```
“连接成功”
“{code:400,errmsg: 'not found controller' }”
“连接断开了”
```

此时你再回到你的 NC 控制台，你将会看到“新的用户到访”字样。

好的，这里解释一下。

客户端，的 connect({ … }) 对 NS 进行连接的建立，在建立完成后，又立即发起一个请求，请求的 channel 名称叫“my_channel_1”，请求的 controller 为“test”，并携带了“abc”作为参数。在发起请求后，NS 收到用户请求，根据用户的 channel 需求派发至频道配置中设定的名为“my_channel_1”的 NC 上，由于这次请求是用户第一次访问该 Channel，所以我们在 Channel 的 OnNewUser 的监听器就捕获到了用户的到访事件，但由于这个 NC 没有任何开发者定义的 Controller，所以将返回 400 错误码（错误码含

义请参考全局代码表）。由于当前用户虽然访问了 Channel，但是依然没有被授权，所以之后还是会被 NS 主动断开连接。

此时，我们不妨多做一次尝试，将 NS 的 Channel 配置文件的“my_channel_1”的“auth”变更为 true，重启 NS，此时再次运行客户端，我们将看到：

“连接成功”
“{code:500,errmsg: 'permission denied' }”
“连接断开了”

而这次 NC 的控制台没有再次弹出“新的用户到访”字样。

所以，我们也就多理解了 NaiveNet 的授权机制，用户在未授权的情况下，可以在短时间内与 NS 建立连接，被强制断开连接前，用户还是有权限访问不需要授权的 NC，那么也就意味着我们需要提供一个不需要授权的 NC 来接待新到访的用户，并根据某些信息确认他的身份合法性，然后给他一个授权。

（不要忘记把 auth 重新写回 false 并重启 NS，否则下面的教程将无法进行）

NC 控制器的添加

刚刚我们创建了 NC，并且也让用户成功的访问了 NC，但是用户的请求返回了 400 的错误代码，这是由于目前开发者还没有为 NC 添加任何 Controller。

现在我们将为 NC 创建一个 Controller，并让用户请求到。

在 NC 的 launch（）前添加如下代码：

```
NaiveNetBox box = new NaiveNetBox();
box.addController(new NaiveNetController("test") {
    @Override
    public NaiveNetResponse onRequest(NaiveNetMessage
msg) {
        try {
            String param = new String(msg.param,"utf-8");
            System.out.println("参数: " + param);
        } catch (UnsupportedEncodingException e) {}
        return null;
    }
});
User.AddBox(box);
```

NC 与 Client 全部都编译运行后，我们将会在 NC 的控制台看到：

“参数： abc”

abc 正是客户端 request 传入的 data！
客户端也收到了

“{code:200,errmsg: 'ok' }”
的回复。
很好，这里需要解释一下。
NaiveNetBox 是作者提供的一个 Controller 的容器，主要目的是为了我们后续业务的控制器打包添加与移除，这种场景非常的多，比如一场“棋牌对弈”，游戏开始前，玩家们可以进行的操作比如：

Box1：“准备/取消准备”、“退出房间” 等操作。

而比赛开始时，则需要

Box2： “落子”，“发起议和”，“……”

除此之外，比如

Box3： “聊天”、“送礼物”、“……”这类无论游戏开始结束都需要的事件。

我们定义了 3 个 Box，就可以在进入房间时，**添加 Box1、Box3**，开始游戏时，**移除 Box1，添加 Box2**，游戏结束时，**再次添加 Box1，移除 Box2**。最后房间解散后，**Box1、Box2、Box3 全部移除**。
这样就很好的进行权限控制。这里可以注意的是，Box 并不需要反复创建，同一类的操作，可以进行局部唯一的创建，因为在 Controller 的事件中，我们能从 NaiveNetMessage msg 中获得事件的来源 User 句柄。即：

```
User user = msg.user;
```

我们再回到 NC 端代码，Box 创建后，添加了很多 Controller，Controller 的构造器接受一个 String 类型参数，代表了 Controller 的名字，对应的是 Client 的 request 的 controller 参数。除了这样定义匿名 Controller 外，你也可以对 Controller 进行继承（extends），这样可以满足更多的编程需要。
然后我们看下最后一行代码，

```
User.AddBox(box);
```

我们创建了 Box，定义了 box 内的 Controller，最后要添加到用户上，而这里并没有使用 User 的实例句柄，而是 User 类名。实际上，User 类与 user 都可以添加 Box，区别是，User.AddBox()是静态方法，作用于所有的 User，而 user.addBox()是 user 的成员方法，添加的 Box 只作用于单例 user。
具体的：

```
User.AddBox(box);  //对所有 User 生效的 Box

user.addBox(box);  //只对当前 user 生效的 Box
```

为用户添加授权

如果你看过 User 实例的成员方法，相信你已经知道如何授权了。只需要执行：

```
msg.user.auth(null);
```

注：该函数接受一个 NaiveNetOnResponse 的实现类，可以理解为事件回调结果，填写 null 代表忽略该回调。NaiveNet 中还有很多事情都是由回调异步完成的。

但是细心地作者仍然要强调一下授权操作。授权操作是一个很危险的行为，因为这将扩大 NS 帮助 Client 访问 NC 的范围。所以授权操作需要建立在一定验证机制后才能执行，而不是访问 NS 的第一件事就是访问一个不需要授权的 NC，直接授权。

在微信小程序（小游戏）开发中，通常使用 wx.login（）方法获得 code，你可以将该 code 使用 ns.request 方式发送到 NC 的特定 Controller 中，然后你在 Controller 内使用后端验证，验证成功后，再进行授权。

授权除了能让用户与 NS 保持连接外，更重要的是能够实现 NaiveNet 非常重要的 Recover 功能来实现网络断线恢复能力。

NC 控制器的回应

我们在此前的开发中，Client 发出的请求，一直都是 NC 在接收请求，并获得参数，而 Client 只得到了一个 code 200 的回复状态，这样的应用场景太单一了。现在我们为 NC 的控制器添加一个回应，使得客户端在发送请求后，能够得到一个开发者自定义的回应。

我们可以观察，控制器中 Override（重写）的 onRequest 方法支持返回一个 NaiveNetResponse 对象，而默认的情况下，我们返回的是 `return null;`。实际上，默认情况下，NaiveNet 为我们默认创建了一个 NaiveNetResponse 对象，就是我们在客户端看到的 code 200。

此时，我们通过 onRequest 方法的参数 NaiveNetMessage msg 获得到该请求的 NaiveNetResponse 句柄就可以自定义回复了。代码如下：

```
NaiveNetResponse res = msg.getResponseHandler();
```

然后为该句柄添加返回的参数正文：

```
res.setContent("你好，欢迎来到 NaiveNet 世界！");
```

注意，因为传输过程中是字节流，为了方便开发者能够有更多的自定义数据内容的能力，所以 setContent 实际上也可以传入的是 byte[] 类型，通常情况

下建议开发者直接传入字符串，因为在 Client 侧默认情况下会自动帮我们进行转码为字符串，用 JSON 字符串作为正文它不香嘛？

最后 `return res;` 客户端再请求后将收到该回应。

Client 添加控制器并做出回应

NaiveNet 强大之处在于，在建立长连接后，客户端与服务器双方形成了完全对等体，这有区别于传统的 HTTP 协议的工作原理（HTTP 协议永远是浏览器主动请求到服务器后服务器才能给出回应）。我们之前成功的让客户端 request 到开发者在 NC 定义的 Controller 并作出回应，实际上 Client 也可以定义 Controller，让 NC 请求，并得到回应。

Client 的控制器也需要由 Box 来打包添加与移除，Box 的创建方法为：

```
let box = ns.createBox();
```

由于前端 JavaScript 的解释器机制原因，前端的控制器定义将稍有变化，添加控制器的标准格式为：

```
box.addController(
    "my_ctrl_1",
    function(res,response){
        //anycode
        //.....

        response();
    }.bind(this)
);
```

box.addController 方法需要传入 2 个参数，第一个参数为 String 类型，是添加 Controller 的名字，第二个参数需要定义一个函数体，即为 Controller 的事件处理函数，为了保持 this 句柄在执行函数体不发生转移，需要对函数尾部添加.bind(this)。

其中，函数体将提供两个参数，第一个参数 res 可获请求时所携带的参数信息。第二个参数 response 将作为回应函数，由于 JavaScript 很多事件需要由异步过程处理完成，因此 NaiveNet 并没有给出和 NC `return null;` 默认回应，因此 Client 开发者必须手动完成 response();的调用，其中 response();支持传入 1 个参数可以作为该控制器的回应内容。

```
response('你好，欢迎来到 NaiveNet 世界！ ');
```

对应的，NC 在请求时将获得该回应的数据。

最后不要忘记将这个 box 添加到 ns 句柄中。

```
ns.addBox(box);
```

NC 请求 Client 的 Controller

Client 定义完 Controller 后，对于 NC 自然可以主动的发起请求，NC 的请求方式为：

使用 user 句柄的 request() 方法完成发送，具体的：

```
user.request(
    "my_ctrl_1",
    "请求参数",
    new NaiveNetOnResponse() {
        @Override
        public void OnComplete(int code, byte[] data) {
            //回调处理函数体
        }}
);
```

request 的方法支持传入 3 个参数，第一个参数 String 类型，是希望请求的 Client 的 Controller 名称。第二个参数为请求携带的参数正文，支持传入 byte[] 类型，但推荐使用 String。第三个参数为 NaiveNetOnResponse 回调处理对象，回调处理对象中 code 代表请求的回应代码，data 为回应的参数内容。在本例中，由于 Client 使用了：

```
response('你好，欢迎来到 NaiveNet 世界！');
```

因此 byte[] data 将是 “你好，欢迎来的 NaiveNet 世界”的 utf-8 编码字节集。

NC 使用 SESSION 能力

项目实际运行时，用户可能需要与多个 NC 产生交互，比如一开始需要访问一个不需要 Auth 的 NC 用来给用户授权，之后将去访问一些需要授权 NC 去处理业务逻辑。

在这期间 NC1 与 NC2 之间可能需要数据的传递，NaiveNet 提供了一个和我们在开发经典 Web 业务时相同的“会话”能力，在任何 NC 的环境内的 user 句柄，都包含一对 SESSION 数据的操作函数，即：

```
user.setSession("KEY", "VALUE".getBytes(), response );
user.getSession("KEY",
    new NaiveNetOnResponse() {
        @Override
        public void OnComplete(int code, byte[] data) {
            //data 就是 KEY 对应的 VALUE 值
        }
    });
```

从函数名我们就可以理解，setSession() 是在为 user 添加一个键值，而 getSession() 是在获取一个键值，设置与获取都是异步的，所以他们最后一个参数都支持传入一个 NaiveNetOnResponse 对象。

SESSION 机制是由 NS 提供的，任何 NC 都可以对 User 进行 SESSION 读存操作，被 NC1 存入的键值，在用户没有彻底离开 NS 前，NC2、NC3……都可以读取到该值，甚至可以修改这个值。

值得注意的是，SESSION 的数据是由 NS 保管，Client 无法获取到。

Client 主动退出一个 NC

Client 借助 NS 可以与任意一个 NC 产生通信，但是实际的业务上，Client 并不需要时刻与所有的 NC 保持通讯，比如一开始由 Auth_NC 验证了用户身份，并成功为 User 授权后，Auth_NC 通过 SESSION 机制保存好 User 的一些信息（比如 主键 ID、头像、昵称等），Auth_NC 就已经不需要再为该 User 提供服务。此时客户端需要在合适的时机释放资源,释放与 NC 的资源不会销毁 SESSION 内容，也不会使用户与 NS 发生连接中断。

客户端在不需要某一个 NC 服务时退出 NC，具体代码为：

```
ns.quitChannel({
    channel:'my_channel_1',
    success:(res)=>{
       //完成退出
    }
})
```

此时，如果你在该 NC 添加了 Quit 事件监听器，同样能获得回调。

提示：合理的释放可以充分利用系统资源。

Client 对断线的恢复能力

用户的客户端在生产环境中是极度不稳定的，比如移动网络与 WiFi 之间的切换等等。不能因为用户的连接中断从而导致业务终止。NaiveNet 提供了断线恢复能力。

在使用断线恢复能力前，必须确保用户已经完成了 Auth，客户端监听到 onBreak 事件回调后，就可以尝试进行网络恢复了。

网络恢复分为两种情况，第一种是在应用程序没有退出时的网络断开：

执行：

```
ns.recover({
    success:(res)=>{
        //成功
```

```
        },
        fail:(res)=>{
            //失败，请查看 res.code 错误码进行处理
        }
    })
```

第一种情况十分简单。

第二种情况是应用程序完全退出的恢复，这种情况会稍微复杂。首先你的客户端要有完备的业务场景恢复处理逻辑，接下来通过 NaiveNet 实现网络恢复。

首先，你需要注册 Client 的 ns 句柄的 onAuth 回调事件，因为在 onAuth 中提供了用于我们恢复的 TOKEN。

```
    ns.onAuth = function(res){
        // res.token 就是我们用于恢复的 TOKEN
    }
```

如果你想让你的应用程序在闪退后也能完成恢复，那么你就需要在 onAuth 回调中获得的 TOKEN 将它存盘保存，因为在闪退的时候，磁盘的内容是依旧存在的。

于是你在应用程序每次启动时，都要读取你曾经保存的 TOKEN（如果他存在，说明很有可能之前发生过闪退），于是你就可以使用 TOKEN 尝试完成网络的恢复工作。

```
    ns.recover({
       ns:'wss://xxxxxxx.xxx',
       token: TOKEN,
       success:(res)=>{},
       fail:(res)=>{}
    })
```

其中，ns 是该用户上次连接的 NS，如果你的 NS 采用了分布式部署模式，上次是 NS1，这次填 NS2 是不能完成恢复的。分布式 NS 的模式下你除了要存盘 TOKEN，还需要存盘 ns 地址，确保两次的地址与 TOKEN 的一致性。

如果能够恢复，则会发生 success 回调。

请注意，请注意，请注意！NaiveNet 只能帮你完成网络恢复，使得你的后端逻辑处理变得很容易，但仍需要你的前后端配合，比如在第一种情况下 Client 发生掉线，掉线期间如果你的 NC 对 Client 发起 request 请求，这期间的请求都将无法抵达 Client,即使 Client 恢复也无法再次触发,因此比如在游戏大厅内，这些消息的丢失就随他去吧，你不需要恢复，但是在游戏中，期间的消息可能非常的关键，这决定了游戏的场景变化，作者建议你的是，在一场游戏中你的 NC 需要做游戏关键数据的有序缓存 list，每一条记录都存在一个有序 ID，当发生断线需要恢复时，Client 提交他本地最后一个消息 ID，然后 NC 会补发期间漏掉的消息，这样就可以恢复游戏进度了。当然也有其他办法完成场景恢复。

如果是第二种情况，你的客户端逻辑处理就更加复杂，需要你自己开动脑筋尝试完成。

Client 优雅的结束连接

连接有建立就会有退出，因为 NaiveNet 的网络恢复机制，这使得开发者需要在用户明确发生退出的时候，结束连接，这样可以更有效的回收资源。如果任由用户直接退出应用程序进程，会使得 NS 仍然保留了一段时间用户句柄。

退出动作非常简单，客户端只需要：

```
ns.close();
```

当执行了该命令后，NS 将彻底释放当前 User 的所有资源，无法再恢复。

一个经典小游戏实战

暂无待补充

高级功能

暂无待补充

成为 NaiveNet 平台一员

暂无待补充

可视化远程控制 NS

暂无待补充

一个标准的增减配流程

暂无待补充

分布式服务的容灾方案

暂无待补充

完全 API 手册

NaiveNetServer（NS）配置手册

NS 一共存在两个配置文件 NaiveNetConfig.json 与 NaiveNetChannel.json。

NaiveNetConfig.json
该配置文件主要用于配置 NS 的启动信息，以 JSON 格式填写。

配置项

属性	类型	必填	描述
SERVER_PORT	int	是	NS 的启动端口号
SSL_JKS_FILEPATH	String	否	如果需要以 SSL,请填写对应的 JKS 文件在服务器上的路径。如果不启动 SSL，则请留空白字符串。
SSL_PASSWORD_FILEPATH	String	否	SSL 对应的 keyStorePass.txt 文件路径。
USER_BREAK_TIMEOUT	int	是	用户非正常掉线的离线判定时间，单位：ms。
USER_QUIT_TIMEOUT	int	是	用户断线后，保留的数据时长，单位：ms。
USER_AUTH_TIMEOUT	int	是	用户供授权的时间，超过该时间将强制断开用户连接，单位：ms。

NaiveNetChannel.json
该配置文件主要用于配置 NS 的 Channel 信息表，以 JSON 数组格式填写。

请注意，由于 NS 支持启动后动态调配 NC 的信息表，因此该配置文件仅供首次启动时生效，如需下次仍然使用手动生效，请删除同级目录中名为“NaiveNetChannel.temp”文件。

数组元素配置项

属性	类型	必填	描述
name	String	是	NS 的名字，对应 Client 的 Channel
ip	String	是	能够让 NS 访问到该 NC 的 IP 地址，如果 NS 与 NC 是同一台服务器，则填写 127.0.0.1，在同一个网段可以填写内网 IP。
port	int	是	该 NC 开放的端口号。
token	String	是	访问 NC 的 token。
auth	Boolean	是	该 NC 是否需要客户端授权后才能访问

NaiveNetChannel（NC）API 手册

Java 版本 SDK

Java 版本通常对应 NaiveNetChannel_x_x.jar 脚本文件。

NaiveNetChannel 类

NaiveNetChannel(int port , String token)
构造器，提供 端口号 与用于 NS 认证该 NC 的 TOEKN ，port 与 token 必须与 NS 的 NaiveNetChannel.json 元素项完全一致。

void launch ()
成员方法，在配置好后启动该服务。

void launch (int MAXTHREAD)
*成员方法，在配置好后启动该服务。MAXTHREAD 支持自定义 Reactor 线程池的最大线程数量，默认情况下是当前服务器 CPU 核数 * 4 个线程。*

void shutdown()
成员方法，关闭服务。

setOnNewUserListener (NaiveNetEvent e)
成员方法，设置新用户访问的监听器，建议在 launch 前配置。

NaiveNetEvent 接口
在 NaiveNetChannel 中发生回调时的接口

void on(User user,byte[] data)
发生回调的成员函数，user 是触发事件的用户句柄，data 是可能携带的扩展参数，2.0 版本暂不存在内容。

User 类
NC 与客户端的交互句柄

void AddBox(NaiveNetBox box)
静态方法，添加 NaiveNetBox，将对所有的 User 生效。

void RemoveBox(NaiveNetBox box)
静态方法，移除对所有 User 生效的 NaiveNetBox。

void addBox(NaiveNetBox box)
成员方法，为当前 User 添加 NaiveNetBox。

void removeBox(NaiveNetBox box)
成员方法，为当前 User 移除 NaiveNetBox。

void setOnQuitListener(NaiveNetBox box)
成员方法，为当前 User 添加发生退出频道事件的监听器。

void setOnBreakListener(NaiveNetBox box)
成员方法，为当前 User 添加发生断线事件的监听器。

void setOnRecoverListener(NaiveNetBox box)
成员方法，为当前 User 添加发生网络恢复事件的监听器。

void request(Sring controller,String/byte[] data , NaiveNetOnResponse onResponse)
成员方法，请求当前 User 的某个控制器。
参数 1 控制器名称，
参数 2 携带的参数，其中支持使用 String 或者 byte[]，
参数 3 当客户端发生回应的回调事件。

void auth(NaiveNetOnResponse onResponse)
成员方法，为当前 User 进行授权。

void getSession(String key , NaiveNetOnResponse onResponse)
成员方法，获取当前用户的一个 SESSION，
参数 1，KEY，
参数 2，获取到 VALUE 的回调。

void setSession(String key,byte[] value,NaiveNetOnResponse onResponse)
成员方法，设置当前用户的一个 SESSION，
参数 1，KEY，
参数 2，VALUE，只支持 byte[]，
参数 3，设置成功的回调。

void getPing(NaiveNetOnResponse onResponse)
成员方法，获取当前用户的网络 ping 值，onResponse 的回调需要进行字符串编码后再转成 int 类型，单位 ms。

NaiveNetOnResponse 接口
请求的响应事件接口

void OnComplete(int code, byte[] data)
发生回调的成员函数，code 是状态码，请参阅全局状态表。data 是可能携带的回应内容，通常需要进行 new String(data); 转化成字符串。

NaiveNetBox 类
请求的响应事件接口

void addController(NaiveNetController controller)
成员方法，为当前 Box 添加 Controller 实体。

void removerController(NaiveNetController controller)
成员方法，为当前 Box 移除 Controller 实体。

NaiveNetController 类
控制器
NaiveNetController(String name)
构造器，提供该控制器的名称。

NaiveNetResponse onRequest(NaiveNetMessage msg)
抽象成员函数（需实现），当发生请求时的处理函数实体，msg 将携带请求的所有信息，请参考 NaiveNetMessage。该函数的返回值接受 NaiveNetResponse 实例，或 null，如果 return null，则系统将自动返回 code 200 的成功通信结果给客户端。如需自定义返回内容请参考 NaiveNetResponse 说明。

NaiveNetMessage 类
控制器的请求参数实体
NaiveNetResponse getResponseHandler()
成员方法，返回用于回复的 NaiveNetResponse 实体。

String controller
成员变量，客户端请求的控制名称。

byte[] param
成员变量，客户端请求携带的参数。

User user
成员变量，客户端 User 句柄。

NaiveNetResponse 类
控制器的用于回复的实体，该实体需要由 NaiveNetMessage 实体获得。
void setContent(String/byte[] content)
成员方法，设置用于回复的内容，支持 String 或者 byte[]。

NaiveNetClient（Client）API 手册

JavaScript 版本 SDK

JavaScript 版本通常对应 NaiveNetClient_x_x.min.js 脚本文件。

获得 NaiveNet 对象（操作句柄）

```
import NaiveNet from 'NaiveNetClient_2_0.min.js';  //引入 NaiveNet SDK
let ns = new NaiveNet();
```

NaiveNet 对象

成员函数说明

成员函数	返回值	描述

connect(Object)	null	与某个 NS 建立连接。
		参数：
		{
		ns:"wss:xxxx.xxx.xx", //某个 NS 的服务地址 需要携带协议头(wss)
		success:function(res){}, //成功的回调
		fail:function(res){}, //失败的回调
		complete:function(res){} //完成后的回调
		}

break(Object)	null	主动临时断线当前连接，该断线可被 recover()恢复，该功能通常在用户 onHide()主动触发，相应的可以在 onShow()使用 recover()进行回复。
		参数：
		{
		eventstatus: boolean, //是否产生 onBreak 回调，可缺省，默认回调，false 时不回调
		}

recover(Object)	null	对当前网络尝试恢复，需要在发生 onBreak 回调，或者主动 break()后使用。
		参数：
		{
		ns:" ", //非闪退情况下可缺省
		token:" ",//用于恢复的口令，由 onAuth 事件获得
		success:function(res){}, //成功的回调
		fail:function(res){}, //失败的回调
		complete:function(res){} //完成后的回调
		}

close()	null	彻底关闭当前连接，关闭后无法进行网络恢复。

request(Object)	null	发起一个请求，并得到回应。
		{
		channel:"", //你的 NC 名称（在 NS 的配置文件中声明的）
		controller:"", //想要访问的 Controller 名称
		data:"", //参数内容 String 或 ArrayBuffer
		success:function(res){ //成功的回调
		/*
		res 结构如下：
		{
		code:200, //状态码
		data: String //NC 的回复 String 内容
		originData: ArrayBuffer //NC 回复的字节集
		}
		*/
		},
		fail:function(res){}, //失败的回调
		complete:function(res){} //完成后的回调
		}

quitChannel(Object)	null	退出一个 NC（但是 NS 的连接不断开）
		{
		channel:"", //你的 NC 名称（在 NS 的配置文件中声明的）
		success:function(res){}, //成功的回调
		fail:function(res){}, //失败的回调
		complete:function(res){} //完成后的回调
		}

createBox()	Object<NaiveNetBox>	创建一个 NaiveNetBox 对象
addBox(Object<NaiveNetBox>)	null	注册一个 NaiveNetBox
removeBox(Object<NaiveNetBox>)	boolean	移除一个 NaiveNetBox
getPing()	int	主动获取当前的 ping 值，单位 ms。

enterChannel(Object)	null	进入一个 NC，通常情况下无需主动调用，NaiveNet 在你 request 时会自动判断并调用。
		{
		channel:"", //你的 NC 名称（在 NS 的配置文件中声明的）
		success:function(res){}, //成功的回调
		fail:function(res){}, //失败的回调
		complete:function(res){} //完成后的回调
		}

事件说明

事件	说明
onBreak	当发生网络断线时发生回调。
onRecover	当发生网络恢复时发生回调

onAuth	当被 NC 授权时发生回调
	回调时携带参数，参数结构为：
	{
	code:200,
	errmsg:'xxxxxx',
	token:'XXXXXXXXX' //可用于闪退恢复的 TOKEN
	}

onPingChange	当 ping 值发生变化时回调 回调时携带参数，参数为： number 单位 ms。

NaiveNetBox 对象

成员函数

成员函数	返回值	描述

addController(
String,
function,
)
	null	向 Box 中添加 Controller。
		第一个参数是 Controller 的名字，
		第二参数填入一个函数定义，该函数需要包含两个参数。例如：
		box.addController(
		'clientCtrl', //控制器名称
		function(res,response){
		console.log(res);
		response('回应的数据');
		}
		)
		其中 res 是来自 NC 的请求参数，结构为：
		{
		controller:", //当前控制器名称
		dataBytes: ArrayBuffer, //请求参数的字节集
		data: String //如果是字符串将被解码在这里输出
		}

removeController(String)	null	移除一个控制器，参数填写的是控制器名称

全局代码表

代码	errmsg	说明
200	ok	成功
400	not found controller	没有找到对应的控制器
401	not found channel	没有找到对应的 NC
402	cannot be established	NS 无法与对应 NC 建立连接，通常情况下是 NC 发生了宕机
403	channel refuse connect	NC 拒绝与 NS 建立连接，通常情况下是 NS 的 NC 配置口令表有误
404	channel not established with server	NS 与 NC 还没有建立连接，通常情况下 NaiveNet 会帮我们自己重试
500	permission denied	客户端在未授权的情况下访问了需要授权的 NC
501	data format error	数据格式异常
502	unknown error	未知错误
503	recover failed	恢复失败，如果接受到该返回码代表当前 NS 彻底无法恢复（这是一个明确的无法恢复的结论）

支持我们

作者希望有朝一日自己的作品能帮助到大家！NaiveNet 的核心功能全部免费使用，如果你使用了 NaiveNet 让你的生活、事业变得越来越好，欢迎你来这里告诉我！同样你的赞助我会铭记于心！