frp

README | 中文文档

Platinum Sponsors

Gold Sponsors

Silver Sponsors

- Sakura Frp - 欢迎点击 "加入我们"

What is frp?

frp is a fast reverse proxy to help you expose a local server behind a NAT or firewall to the Internet. As of now, it supports **TCP** and **UDP**, as well as **HTTP** and **HTTPS** protocols, where requests can be forwarded to internal services by domain name.

frp also has a P2P connect mode.

Table of Contents

Development Status

frp is under development. Try the latest release version in the `master` branch, or use the `dev` branch for the version in development.

We are working on v2 version and trying to do some code refactor and improvements. It won't be compatible with v1.

We will switch v0 to v1 at the right time and only accept bug fixes and improvements instead of big feature requirements.

Architecture

Example Usage

Firstly, download the latest programs from Release page according to your operating system and architecture.

Put `frps` and `frps.ini` onto your server A with public IP.

Put `frpc` and `frpc.ini` onto your server B in LAN (that can't be connected from public Internet).

Access your computer in LAN by SSH

1. Modify `frps.ini` on server A and set the `bind_port` to be connected to frp clients:

```
# frps.ini
[common]
bind_port = 7000
```

2. Start `frps` on server A:

```
./frps -c ./frps.ini
```

3. On server B, modify `frpc.ini` to put in your `frps` server public IP as `server_addr` field:

```
# frpc.ini
[common]
server_addr = x.x.x.x
server_port = 7000

[ssh]
type = tcp
local_ip = 127.0.0.1
local_port = 22
remote_port = 6000
```

Note that `local_port` (listened on client) and `remote_port` (exposed on server) are for traffic goes in/out the frp system, whereas `server_port` is used between frps.

4. Start `frpc` on server B:

`./frpc -c ./frpc.ini`

5. From another machine, SSH to server B like this (assuming that username is `test`):

`ssh -oPort=6000 test@x.x.x.x`

Visit your web service in LAN by custom domains

Sometimes we want to expose a local web service behind a NAT network to others for testing with your own domain name and unfortunately we can't resolve a domain name to a local IP.

However, we can expose an HTTP(S) service using frp.

1. Modify `frps.ini`, set the vhost HTTP port to 8080:

```
# frps.ini
[common]
bind_port = 7000
vhost_http_port = 8080
```

2. Start `frps`:

`./frps -c ./frps.ini`

3. Modify `frpc.ini` and set `server_addr` to the IP address of the remote frps server. The `local_port` is the port of your web service:

```
# frpc.ini
[common]
server_addr = x.x.x.x
server_port = 7000

[web]
type = http
local_port = 80
custom_domains = www.example.com
```

4. Start `frpc`:

`./frpc -c ./frpc.ini`

5. Resolve A record of `www.example.com` to the public IP of the remote frps server or CNAME record to your origin domain.

6. Now visit your local web service using url `http://www.example.com:8080`.

Forward DNS query request

1. Modify `frps.ini`:

```
# frps.ini
[common]
bind_port = 7000
```

2. Start `frps`:

```
./frps -c ./frps.ini
```

3. Modify `frpc.ini` and set `server_addr` to the IP address of the remote frps server, forward DNS query request to Google Public DNS server `8.8.8.8:53`:

```
# frpc.ini
[common]
server_addr = x.x.x.x
server_port = 7000

[dns]
type = udp
local_ip = 8.8.8.8
local_port = 53
remote_port = 6000
```

4. Start frpc:

```
./frpc -c ./frpc.ini
```

5. Test DNS resolution using `dig` command:

```
dig @x.x.x.x -p 6000 www.google.com
```

Forward Unix domain socket

Expose a Unix domain socket (e.g. the Docker daemon socket) as TCP.

Configure `frps` same as above.

1. Start `frpc` with configuration:

```
# frpc.ini
[common]
server_addr = x.x.x.x
server_port = 7000

[unix_domain_socket]
type = tcp
remote_port = 6000
plugin = unix_domain_socket
plugin_unix_path = /var/run/docker.sock
```

2. Test: Get Docker version using `curl`:

```
curl http://x.x.x.x:6000/version
```

Expose a simple HTTP file server

Browser your files stored in the LAN, from public Internet.

Configure `frps` same as above.

1. Start `frpc` with configuration:

```
# frpc.ini
[common]
server_addr = x.x.x.x
server_port = 7000

[test_static_file]
type = tcp
remote_port = 6000
plugin = static_file
plugin_local_path = /tmp/files
plugin_strip_prefix = static
plugin_http_user = abc
plugin_http_passwd = abc
```

2. Visit `http://x.x.x.x:6000/static/` from your browser and specify correct user and password to view files in `/tmp/files` on the `frpc` machine.

Enable HTTPS for local HTTP(S) service

You may substitute `https2https` for the plugin, and point the `plugin_local_addr` to a HTTPS endpoint.

1. Start `frpc` with configuration:

```
# frpc.ini
[common]
server_addr = x.x.x.x
server_port = 7000

[test_https2http]
type = https
custom_domains = test.example.com

plugin = https2http
plugin_local_addr = 127.0.0.1:80
plugin_crt_path = ./server.crt
plugin_key_path = ./server.key
plugin_host_header_rewrite = 127.0.0.1
plugin_header_X-From-Where = frp
```

2. Visit `https://test.example.com` .

Expose your service privately

Some services will be at risk if exposed directly to the public network. With **STCP** (secret TCP) mode, a preshared key is needed to access the service from another client.

Configure `frps` same as above.

1. Start `frpc` on machine B with the following config. This example is for exposing the SSH service (port 22), and note the `sk` field for the preshared key, and that the `remote_port` field is removed here:

```
# frpc.ini
[common]
server_addr = x.x.x.x
server_port = 7000

[secret_ssh]
type = stcp
sk = abcdefg
local_ip = 127.0.0.1
local_port = 22
```

2. Start another `frpc` (typically on another machine C) with the following config to access the SSH service with a security key (`sk` field):

```
# frpc.ini
[common]
server_addr = x.x.x.x
server_port = 7000

[secret_ssh_visitor]
type = stcp
role = visitor
server_name = secret_ssh
sk = abcdefg
bind_addr = 127.0.0.1
bind_port = 6000
```

3. On machine C, connect to SSH on machine B, using this command:

```
ssh -oPort=6000 127.0.0.1
```

P2P Mode

xtcp is designed for transmitting large amounts of data directly between clients. A frps server is still needed, as P2P here only refers the actual data transmission.

Note it can't penetrate all types of NAT devices. You might want to fallback to **stcp** if **xtcp** doesn't work.

1. In `frps.ini` configure a UDP port for xtcp:

```
# frps.ini
bind_udp_port = 7001
```

2. Start `frpc` on machine B, expose the SSH port. Note that `remote_port` field is removed:

```
# frpc.ini
[common]
```

```
server_addr = x.x.x.x
server_port = 7000

[p2p_ssh]
type = xtcp
sk = abcdefg
local_ip = 127.0.0.1
local_port = 22
```

3. Start another `frpc` (typically on another machine C) with the config to connect to SSH using P2P mode:

```
# frpc.ini
[common]
server_addr = x.x.x.x
server_port = 7000

[p2p_ssh_visitor]
type = xtcp
role = visitor
server_name = p2p_ssh
sk = abcdefg
bind_addr = 127.0.0.1
bind_port = 6000
```

4. On machine C, connect to SSH on machine B, using this command:

```
ssh -oPort=6000 127.0.0.1
```

Features

Configuration Files

Read the full example configuration files to find out even more features not described here.

Full configuration file for frps (Server)

Full configuration file for frpc (Client)

Using Environment Variables

Environment variables can be referenced in the configuration file, using Go's standard format:

```
# frpc.ini
[common]
server_addr = {{ .Envs.FRP_SERVER_ADDR }}
server_port = 7000

[ssh]
type = tcp
local_ip = 127.0.0.1
local_port = 22
remote_port = {{ .Envs.FRP_SSH_REMOTE_PORT }}
```

With the config above, variables can be passed into `frpc` program like this:

```
export FRP_SERVER_ADDR="x.x.x.x"
export FRP_SSH_REMOTE_PORT="6000"
./frpc -c ./frpc.ini
```

`frpc` will render configuration file template using OS environment variables. Remember to prefix your reference with `.Envs`.

Split Configures Into Different Files

You can split multiple proxy configs into different files and include them in the main file.

```
# frpc.ini
[common]
server_addr = x.x.x.x
server_port = 7000
includes=./confd/*.ini
```

```
# ./confd/test.ini
[ssh]
type = tcp
local_ip = 127.0.0.1
local_port = 22
remote_port = 6000
```

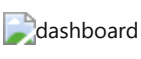
Dashboard

Check frp's status and proxies' statistics information by Dashboard.

Configure a port for dashboard to enable this feature:

```
[common]
dashboard_port = 7500
# dashboard's username and password are both optional
dashboard_user = admin
dashboard_pwd = admin
```

Then visit `http://[server_addr]:7500` to see the dashboard, with username and password both being `admin`.

 dashboard

Admin UI

The Admin UI helps you check and manage frpc's configuration.

Configure an address for admin UI to enable this feature:

```
[common]
admin_addr = 127.0.0.1
admin_port = 7400
admin_user = admin
admin_pwd = admin
```

Then visit `http://127.0.0.1:7400` to see admin UI, with username and password both being `admin`.

Monitor

When dashboard is enabled, frps will save monitor data in cache. It will be cleared after process restart.

Prometheus is also supported.

Prometheus

Enable dashboard first, then configure `enable_prometheus = true` in `frps.ini`.

`http://{dashboard_addr}/metrics` will provide prometheus monitor data.

Authenticating the Client

There are 2 authentication methods to authenticate frpc with frps.

You can decide which one to use by configuring `authentication_method` under `[common]` in `frpc.ini` and `frps.ini`.

Configuring `authenticate_heartbeats = true` under `[common]` will use the configured authentication method to add and validate authentication on every heartbeat between frpc and frps.

Configuring `authenticate_new_work_conns = true` under `[common]` will do the same for every new work connection between frpc and frps.

Token Authentication

When specifying `authentication_method = token` under `[common]` in `frpc.ini` and `frps.ini` - token based authentication will be used.

Make sure to specify the same `token` in the `[common]` section in `frps.ini` and `frpc.ini` for frpc to pass frps validation

OIDC Authentication

When specifying `authentication_method = oidc` under `[common]` in `frpc.ini` and `frps.ini` - OIDC based authentication will be used.

OIDC stands for OpenID Connect, and the flow used is called Client Credentials Grant.

To use this authentication type - configure `frpc.ini` and `frps.ini` as follows:

```
# frps.ini
[common]
authentication_method = oidc
oidc_issuer = https://example-oidc-issuer.com/
oidc_audience = https://oidc-audience.com/.default
```

```
# frpc.ini
[common]
authentication_method = oidc
oidc_client_id = 98692467-37de-409a-9fac-bb2585826f18 # Replace with OIDC client ID
oidc_client_secret = oidc_secret
oidc_audience = https://oidc-audience.com/.default
oidc_token_endpoint_url = https://example-oidc-endpoint.com/oauth2/v2.0/token
```

Encryption and Compression

The features are off by default. You can turn on encryption and/or compression:

```
# frpc.ini
[ssh]
type = tcp
local_port = 22
remote_port = 6000
use_encryption = true
use_compression = true
```

TLS

frp supports the TLS protocol between `frpc` and `frps` since v0.25.0.

For port multiplexing, frp sends a first byte `0x17` to dial a TLS connection.

Configure `tls_enable = true` in the `[common]` section to `frpc.ini` to enable this feature.

To **enforce** `frps` to only accept TLS connections - configure `tls_only = true` in the `[common]` section in `frps.ini`. **This is optional.**

`frpc` TLS settings (under the `[common]` section):

```
tls_enable = true
tls_cert_file = certificate.crt
tls_key_file = certificate.key
tls_trusted_ca_file = ca.crt
```

`frps` TLS settings (under the `[common]` section):

```
tls_only = true
tls_enable = true
tls_cert_file = certificate.crt
tls_key_file = certificate.key
tls_trusted_ca_file = ca.crt
```

You will need **a root CA cert** and **at least one SSL/TLS certificate**. It **can** be self-signed or regular (such as Let's Encrypt or another SSL/TLS certificate provider).

If you using `frp` via IP address and not hostname, make sure to set the appropriate IP address in the Subject Alternative Name (SAN) area when generating SSL/TLS Certificates.

Given an example:

- Prepare openssl config file. It exists at `/etc/pki/tls/openssl.cnf` in Linux System and `/System/Library/OpenSSL/openssl.cnf` in MacOS, and you can copy it to current path, like `cp /etc/pki/tls/openssl.cnf ./my-openssl.cnf`. If not, you can build it by yourself, like:

```
cat > my-openssl.cnf << EOF
[ ca ]
default_ca = CA_default
[ CA_default ]
x509_extensions = usr_cert
[ req ]
default_bits        = 2048
default_md          = sha256
default_keyfile     = privkey.pem
distinguished_name  = req_distinguished_name
attributes          = req_attributes
x509_extensions     = v3_ca
string_mask         = utf8only
[ req_distinguished_name ]
[ req_attributes ]
[ usr_cert ]
basicConstraints       = CA:FALSE
nsComment              = "OpenSSL Generated Certificate"
subjectKeyIdentifier   = hash
authorityKeyIdentifier = keyid,issuer
[ v3_ca ]
subjectKeyIdentifier   = hash
authorityKeyIdentifier = keyid:always,issuer
basicConstraints       = CA:true
EOF
```

- build ca certificates:

```
openssl genrsa -out ca.key 2048
openssl req -x509 -new -nodes -key ca.key -subj "/CN=example.ca.com" -days 5000 -out
ca.crt
```

- build frps certificates:

```
openssl genrsa -out server.key 2048

openssl req -new -sha256 -key server.key \
    -subj "/C=XX/ST=DEFAULT/L=DEFAULT/O=DEFAULT/CN=server.com" \
    -reqexts SAN \
    -config <(cat my-openssl.cnf <(printf
"\n[SAN]\nsubjectAltName=DNS:localhost,IP:127.0.0.1,DNS:example.server.com")) \
    -out server.csr
```

```
openssl x509 -req -days 365 \
    -in server.csr -CA ca.crt -CAkey ca.key -CAcreateserial \
    -extfile <(printf
"subjectAltName=DNS:localhost,IP:127.0.0.1,DNS:example.server.com") \
    -out server.crt
```

- build frpc certificates:

```
openssl genrsa -out client.key 2048
openssl req -new -sha256 -key client.key \
    -subj "/C=XX/ST=DEFAULT/L=DEFAULT/O=DEFAULT/CN=client.com" \
    -reqexts SAN \
    -config <(cat my-openssl.cnf <(printf
"\n[SAN]\nsubjectAltName=DNS:client.com,DNS:example.client.com")) \
    -out client.csr

openssl x509 -req -days 365 \
    -in client.csr -CA ca.crt -CAkey ca.key -CAcreateserial \
    -extfile <(printf "subjectAltName=DNS:client.com,DNS:example.client.com") \
    -out client.crt
```

Hot-Reloading frpc configuration

The `admin_addr` and `admin_port` fields are required for enabling HTTP API:

```ini
# frpc.ini
[common]
admin_addr = 127.0.0.1
admin_port = 7400
```

Then run command `frpc reload -c ./frpc.ini` and wait for about 10 seconds to let `frpc` create or update or remove proxies.

Note that parameters in [common] section won't be modified except 'start'.

You can run command `frpc verify -c ./frpc.ini` before reloading to check if there are config errors.

Get proxy status from client

Use `frpc status -c ./frpc.ini` to get status of all proxies. The `admin_addr` and `admin_port` fields are required for enabling HTTP API.

Only allowing certain ports on the server

`allow_ports` in `frps.ini` is used to avoid abuse of ports:

```ini
# frps.ini
[common]
allow_ports = 2000-3000,3001,3003,4000-50000
```

`allow_ports` consists of specific ports or port ranges (lowest port number, dash `-`, highest port number), separated by comma `,`.

Port Reuse

`vhost_http_port` and `vhost_https_port` in frps can use same port with `bind_port`. frps will detect the connection's protocol and handle it correspondingly.

We would like to try to allow multiple proxies bind a same remote port with different protocols in the future.

Bandwidth Limit

For Each Proxy

```
# frpc.ini
[ssh]
type = tcp
local_port = 22
remote_port = 6000
bandwidth_limit = 1MB
```

Set `bandwidth_limit` in each proxy's configure to enable this feature. Supported units are `MB` and `KB`.

TCP Stream Multiplexing

frp supports tcp stream multiplexing since v0.10.0 like HTTP2 Multiplexing, in which case all logic connections to the same frpc are multiplexed into the same TCP connection.

You can disable this feature by modify `frps.ini` and `frpc.ini`:

```
# frps.ini and frpc.ini, must be same
[common]
tcp_mux = false
```

Support KCP Protocol

KCP is a fast and reliable protocol that can achieve the transmission effect of a reduction of the average latency by 30% to 40% and reduction of the maximum delay by a factor of three, at the cost of 10% to 20% more bandwidth wasted than TCP.

KCP mode uses UDP as the underlying transport. Using KCP in frp:

1. Enable KCP in frps:

```
# frps.ini
[common]
bind_port = 7000
# Specify a UDP port for KCP.
kcp_bind_port = 7000
```

The `kcp_bind_port` number can be the same number as `bind_port`, since `bind_port` field specifies a TCP port.

2. Configure `frpc.ini` to use KCP to connect to frps:

```
# frpc.ini
[common]
server_addr = x.x.x.x
# Same as the 'kcp_bind_port' in frps.ini
server_port = 7000
protocol = kcp
```

Connection Pooling

By default, frps creates a new frpc connection to the backend service upon a user request. With connection pooling, frps keeps a certain number of pre-established connections, reducing the time needed to establish a connection.

This feature is suitable for a large number of short connections.

1. Configure the limit of pool count each proxy can use in `frps.ini`:

```
# frps.ini
[common]
max_pool_count = 5
```

2. Enable and specify the number of connection pool:

```
# frpc.ini
[common]
pool_count = 1
```

Load balancing

Load balancing is supported by `group`.

This feature is only available for types `tcp`, `http`, `tcpmux` now.

```
# frpc.ini
[test1]
type = tcp
local_port = 8080
remote_port = 80
group = web
group_key = 123

[test2]
type = tcp
local_port = 8081
remote_port = 80
group = web
group_key = 123
```

`group_key` is used for authentication.

Connections to port 80 will be dispatched to proxies in the same group randomly.

For type `tcp`, `remote_port` in the same group should be the same.

For type `http`, `custom_domains`, `subdomain`, `locations` should be the same.

Service Health Check

Health check feature can help you achieve high availability with load balancing.

Add `health_check_type = tcp` or `health_check_type = http` to enable health check.

With health check type **tcp**, the service port will be pinged (TCPing):

```
# frpc.ini
[test1]
type = tcp
local_port = 22
remote_port = 6000
# Enable TCP health check
health_check_type = tcp
# TCPing timeout seconds
health_check_timeout_s = 3
# If health check failed 3 times in a row, the proxy will be removed from frps
health_check_max_failed = 3
# A health check every 10 seconds
health_check_interval_s = 10
```

With health check type **http**, an HTTP request will be sent to the service and an HTTP 2xx OK response is expected:

```
# frpc.ini
[web]
type = http
local_ip = 127.0.0.1
local_port = 80
custom_domains = test.example.com
# Enable HTTP health check
health_check_type = http
# frpc will send a GET request to '/status'
# and expect an HTTP 2xx OK response
health_check_url = /status
health_check_timeout_s = 3
health_check_max_failed = 3
health_check_interval_s = 10
```

Rewriting the HTTP Host Header

By default frp does not modify the tunneled HTTP requests at all as it's a byte-for-byte copy.

However, speaking of web servers and HTTP requests, your web server might rely on the `Host` HTTP header to determine the website to be accessed. frp can rewrite the `Host` header when forwarding the HTTP requests, with the `host_header_rewrite` field:

```
# frpc.ini
[web]
type = http
local_port = 80
custom_domains = test.example.com
host_header_rewrite = dev.example.com
```

The HTTP request will have the the `Host` header rewritten to `Host: dev.example.com` when it reaches the actual web server, although the request from the browser probably has `Host: test.example.com`.

Setting other HTTP Headers

Similar to `Host`, You can override other HTTP request headers with proxy type `http`.

```
# frpc.ini
[web]
type = http
local_port = 80
custom_domains = test.example.com
host_header_rewrite = dev.example.com
header_X-From-Where = frp
```

Note that parameter(s) prefixed with `header_` will be added to HTTP request headers.

In this example, it will set header `X-From-Where: frp` in the HTTP request.

Get Real IP

HTTP X-Forwarded-For

This feature is for http proxy only.

You can get user's real IP from HTTP request headers `X-Forwarded-For`.

Proxy Protocol

frp supports Proxy Protocol to send user's real IP to local services. It support all types except UDP.

Here is an example for https service:

```
# frpc.ini
[web]
type = https
local_port = 443
custom_domains = test.example.com

# now v1 and v2 are supported
proxy_protocol_version = v2
```

You can enable Proxy Protocol support in nginx to expose user's real IP in HTTP header `X-Real-IP`, and then read `X-Real-IP` header in your web service for the real IP.

Require HTTP Basic Auth (Password) for Web Services

Anyone who can guess your tunnel URL can access your local web server unless you protect it with a password.

This enforces HTTP Basic Auth on all requests with the username and password specified in frpc's configure file.

It can only be enabled when proxy type is http.

```
# frpc.ini
[web]
type = http
local_port = 80
custom_domains = test.example.com
http_user = abc
http_pwd = abc
```

Visit `http://test.example.com` in the browser and now you are prompted to enter the username and password.

Custom Subdomain Names

It is convenient to use `subdomain` configure for http and https types when many people share one frps server.

```
# frps.ini
subdomain_host = frps.com
```

Resolve `*.frps.com` to the frps server's IP. This is usually called a Wildcard DNS record.

```
# frpc.ini
[web]
type = http
local_port = 80
subdomain = test
```

Now you can visit your web service on `test.frps.com`.

Note that if `subdomain_host` is not empty, `custom_domains` should not be the subdomain of `subdomain_host`.

URL Routing

frp supports forwarding HTTP requests to different backend web services by url routing.

`locations` specifies the prefix of URL used for routing. frps first searches for the most specific prefix location given by literal strings regardless of the listed order.

```
# frpc.ini
[web01]
type = http
local_port = 80
custom_domains = web.example.com
```

```
locations = /

[web02]
type = http
local_port = 81
custom_domains = web.example.com
locations = /news,/about
```

HTTP requests with URL prefix `/news` or `/about` will be forwarded to **web02** and other requests to **web01**.

TCP Port Multiplexing

frp supports receiving TCP sockets directed to different proxies on a single port on frps, similar to `vhost_http_port` and `vhost_https_port`.

The only supported TCP port multiplexing method available at the moment is `httpconnect` - HTTP CONNECT tunnel.

When setting `tcpmux_httpconnect_port` to anything other than 0 in frps under `[common]`, frps will listen on this port for HTTP CONNECT requests.

The host of the HTTP CONNECT request will be used to match the proxy in frps. Proxy hosts can be configured in frpc by configuring `custom_domain` and / or `subdomain` under `type = tcpmux` proxies, when `multiplexer = httpconnect`.

For example:

```
# frps.ini
[common]
bind_port = 7000
tcpmux_httpconnect_port = 1337
```

```
# frpc.ini
[common]
server_addr = x.x.x.x
server_port = 7000

[proxy1]
type = tcpmux
multiplexer = httpconnect
custom_domains = test1
local_port = 80

[proxy2]
type = tcpmux
multiplexer = httpconnect
custom_domains = test2
local_port = 8080
```

In the above configuration - frps can be contacted on port 1337 with a HTTP CONNECT header such as:

```
CONNECT test1 HTTP/1.1\r\n\r\n
```

and the connection will be routed to `proxy1`.

Connecting to frps via HTTP PROXY

frpc can connect to frps using HTTP proxy if you set OS environment variable `HTTP_PROXY`, or if `http_proxy` is set in frpc.ini file.

It only works when protocol is tcp.

```
# frpc.ini
[common]
server_addr = x.x.x.x
server_port = 7000
http_proxy = http://user:pwd@192.168.1.128:8080
```

Range ports mapping

Proxy with names that start with `range:` will support mapping range ports.

```
# frpc.ini
[range:test_tcp]
type = tcp
local_ip = 127.0.0.1
local_port = 6000-6006,6007
remote_port = 6000-6006,6007
```

frpc will generate 8 proxies like `test_tcp_0`, `test_tcp_1`, ..., `test_tcp_7`.

Client Plugins

frpc only forwards requests to local TCP or UDP ports by default.

Plugins are used for providing rich features. There are built-in plugins such as `unix_domain_socket`, `http_proxy`, `socks5`, `static_file`, `http2https`, `https2http`, `https2https` and you can see example usage.

Specify which plugin to use with the `plugin` parameter. Configuration parameters of plugin should be started with `plugin_`. `local_ip` and `local_port` are not used for plugin.

Using plugin **http_proxy**:

```
# frpc.ini
[http_proxy]
type = tcp
remote_port = 6000
plugin = http_proxy
plugin_http_user = abc
plugin_http_passwd = abc
```

`plugin_http_user` and `plugin_http_passwd` are configuration parameters used in `http_proxy` plugin.

Server Manage Plugins

Read the document.

Find more plugins in gofrp/plugin.

Development Plan

- Log HTTP request information in frps.

Contributing

Interested in getting involved? We would like to help you!

- Take a look at our issues list and consider sending a Pull Request to **dev branch**.
- If you want to add a new feature, please create an issue first to describe the new feature, as well as the implementation approach. Once a proposal is accepted, create an implementation of the new features and submit it as a pull request.
- Sorry for my poor English. Improvements for this document are welcome, even some typo fixes.
- If you have great ideas, send an email to fatedier@gmail.com.

Note: We prefer you to give your advise in issues, so others with a same question can search it quickly and we don't need to answer them repeatedly.

Donation

If frp helps you a lot, you can support us by:

GitHub Sponsors

Support us by Github Sponsors.

You can have your company's logo placed on README file of this project.

PayPal

Donate money by PayPal to my account **fatedier@gmail.com**.