

mod_jk 分析

1 mod_jk 模块的总体功能

由于 tomcat 的 HTTP 处理部分都由 Java 所写（5.5.12 版本以后出现了 native 库，用以提高其 I/O 和 SSL 的性能[1]），在高并发的情况下负载较高。而 apache 对于静态文件的处理能力比 tomcat 强，所以 tomcat 开发组开发了与 apache 结合使用的 mod_jk 模块。该协议由 apache 作请求代理，将 HTTP 协议的请求转化为 AJP 协议包，并传给后端的 tomcat。mod_jk 和 apache 现在普遍使用 AJP1.3 协议[2]。它是一个二进制格式的协议，比字符格式的 HTTP 协议解析速度要快。

除了性能的提升，mod_jk 另外的一个作用可以实现 apache 与 tomcat 一对多的对应，使后端 tomcat 负载均衡。mod_jk 也提供 apache 与 tomcat 链接情况的监控。

mod_jk 模块的典型工作流程是这样的：一个 HTTP 请求过来，mod_jk 模块根据其 URI 选择合适的 worker 来进行处理。如果是 lb_worker（负载均衡的 worker），就再根据各种条件选择后台合适的 ajp_worker（处理 AJP 协议的 worker）。ajp_worker 将 HTTP 协议的包，组装成 AJP 协议格式的包，然后选取一条空闲的连接，发送给后台的 tomcat 服务器。等到后台将数据发送过来时，接收并解析 AJP 协议，重新组装成 HTTP 协议，然后把结果发送给客户端。

2 mod_jk 模块的框架

2.1 线程

从宏观上来讲，mod_jk 由一个 watchdog 线程和一组 worker 线程(进程)组成。watchdog 线程是在 apache 内部新创建的线程，它是一个维护线程。每隔 JkWatchdogInterval 的时间（当然，所有 worker 线程也有一个统一的 worker.maintain 时间，JkWatchdogInterval 应该至少大于 worker.maintain），它会扫描所有 worker 线程。watchdog 线程会检查每个 worker 线程的空闲链接、负载情况、与后端的链接情况，并使共享内存同步。worker 线程是就是一些 ajp13, ajp14, jni, lb 或 status 类型的线程，负责所有实际的工作。

在 mod_jk 中，线程内（外）的同步均是通过线程锁（pthread_mutex_lock）来实现的。而进程之间的全局同步，是用文件记录锁（flock 或 fcntl）来实现。进程间的数据共享是用这样做的：进程启动时，创建一个 JkShmSize 大小的文件，然后 mmap 到内存，由于各进程 mmap 到内存的是相同的镜像，所以可以实现数据的共享，但是写入共享内存时，要做好互斥。

由于 apache 的基本处理方式（prefork 和 worker 模式）就是一个线程/进程负责一个连接，所以 mod_jk 各线程中对于网络 IO 处理都是阻塞的。

2.2 worker 对象

从具体的 worker 对象的角度来讲，mod_jk 由 ajp_worker、jni_worker、lb_worker 和 status_worker 组成。这些对象参考了设计模式中 factory 的模型。每类 worker 都有生产 workers 的 factory。

在 mod_jk 中，其中的 worker 主要在 worker.list 中列出。其中，lb_worker 可以含有 balance_workers，以 lb_sub_worker 的形式存储于 lb_worker 中。lb_sub_worker 可以是各

种类型的 `ajp_worker`。所以真正工作的 `ajp_worker` 既可以“单干”，也可以由 `lb_worker` 来分配任务。这主要取决于 URI 到底映射到哪个 worker 上以及该 worker 是否在 `worker.list` 配置。

`lb_worker`，`lb_sub_worker` 和 `ajp_worker` 一些配置信息都位于其结构体中，而状态信息或在运行中可变的参数则位于共享内存中的对应结构体中，当然也并不绝对，有些参数是冗余的。

从正在运行的某个线程的角度上来讲，`ajp_worker` 就是对应了一个线程。

3 从 HTTP 到 AJP 的处理流程

由于 `mod_jk` 模块是 apache 的处理模块，本节主要是讲述 `mod_jk` 模块从客户端到后端服务器的处理流程。中间会涉及一些 apache 模块的一些结构。

3.1 `mod_jk` 模块在 apache 中的定义

3.1.1 `mod_jk` 定义

```
/* 这是 jk 模块的主要定义结构体*/
```

```
module AP_MODULE_DECLARE_DATA jk_module = {
    STANDARD20_MODULE_STUFF,
    NULL,                               /* dir config creator */
    NULL,                               /* dir merger --- default is to override */
    create_jk_config,                  /*创建 jk 模块的配置结构体*/
    merge_jk_config,                  /* 初始化及合并 jk 模块的配置结构体*/
    jk_cmds,                           /* 所有在 apahce 中的命令及操作函数 */
    jk_register_hooks                  /* 具体的操作函数处理钩子 */
};
```

3.1.2 `mod_jk` 模块的主要处理函数

```
/* mod_jk 将各处理函数挂到相应的钩子上，钩子实际上就是一些函数指针。针对某个 HTTP 请求，这些函数会自上而下执行。*/
```

```
static void jk_register_hooks(apr_pool_t * p)
```

```
{
```

```
    /* 该函数在 apache 读入配置后运行，用以初始化全局互斥锁，jk 日志，全局变量的初始化，以及读入 workers.properties、uriworkermap.properties 文件，初始化各 worker 的属性，建立 worker 名称到 worker 结构体的映射，uri 到 worker 的映射*/
```

```
    ap_hook_post_config(jk_post_config, NULL, NULL, APR_HOOK_MIDDLE);
```

```
    /*该函数在 apache 主进程 fork 工作子进程后做的初始化工作，主要包括初始化进程内互斥锁，开启维护线程，读入共享内存*/
```

```

    ap_hook_child_init(jk_child_init, NULL, NULL, APR_HOOK_MIDDLE);
    /* 将 mod_jk 的 URI 到真实 URI，然后 URI 到 worker 的映射翻译过程，用以找到该 URI
    相应的 worker_name */
    ap_hook_translate_name(jk_translate, NULL, NULL, APR_HOOK_MIDDLE);
#if (MODULE_MAGIC_NUMBER_MAJOR > 20010808)
    /* 绑定那些 alias_dir 的 URI 到 mod_jk 的配置，并找到相应 worker_name */
    ap_hook_map_to_storage(jk_map_to_storage, NULL, NULL, APR_HOOK_MIDDLE);
    /* 这是 mod_jk 的主要处理函数 */
    ap_hook_handler(jk_handler, NULL, NULL, APR_HOOK_MIDDLE);
#endif
}

```

3.2 jk_handler 函数

jk_handler 函数是 mod_jk 的主要处理函数，它负责将 HTTP 请求从 apache 发到 tomcat，再从 tomcat 接受数据，并返回给客户。

它的处理过程如下：

1. 根据前面得到的 worker_name，找到相应的 jk_worker（其结构体见 4.4）。
2. 初始化服务结构体 jk_ws_service，主要是针对 HTTP 服务端的一些设置及函数（服务的结构体见 4.5），与 AJP 协议关系不大。
3. 调用 jk_worker 的 get_endpoint 函数，获取具体的 jk_endpoint 结构体（函数见 3.3.1 和 3.4.1，jk_endpoint 的结构体定义见 4.4）。
4. 调用上述 jk_endpoint 的 service 函数。
5. 调用上述 jk_endpoint 的 done 函数，结束与 tomcat 的 AJP 连接。
6. 根据函数的结果处理各种状态。
7. 释放资源。

3.3 lb_worker 的处理函数

3.3.1 lb_worker 的 get_endpoint

初始化 lb_worker 的 endpoint 结构体，挂上其 service 及 done 函数。

3.3.2 lb_worker 的 service 函数

它的处理过程如下：

1. 获取共享内存中记录的 worker 状态。
2. 如果需要绑定 sessionID，获取 sessionID。
3. 调用 get_most_suitable_worker 函数，获取相应的 lb_sub_worker（也即

ajp_worker)。

4. 调用 ajp_worker 的 get_endpoint 函数。
5. 调用上述 endpoint 的 service 函数。
6. 调用上述 endpoint 的 done 函数。
7. 根据 service 的返回结果，更新各种记录状态。

3.3.3 get_most_suitable_worker 函数

它的处理过程如下：

1. 如果需要绑定 sessionID，就根据 sessionID 找到工作正常的最佳 worker。
2. 如果找不到，则调用 find_best_worker 函数。其处理过程如下：
 - i. 又调用 find_best_byvalue 函数。其处理过程如下：按照 Round Robin 的方式在本 lb_worker 中找到第一个不在错误状态、也没有停止、没有 disabled、也不 busy 的 lb_sub_worker。
 - ii. 如果 find_best_byvalue 返回错误，说明本 lb_worker 的 lb_sub_worker 都处于非正常工作状态，需要调用 find_failover_worker 函数，通过 redirect、route、domain 指令来进行查找[3]。

3.4 ajp_worker 的处理函数

3.4.1 ajp_worker 的 get_endpoint 函数

它的主要功能是：找到与后端 tomcat 的一个空闲连接。

3.4.2 ajp_worker 的 ajp_service 函数

该函数主要分为 ajp_send_request 和 and ajp_get_reply 两个函数。

它的处理过程如下：

1. 获取共享内存中的状态。
2. 分配 AJP 请求包、恢复包、POST 包的内存。
3. 调用 ajp_marshall_into_msgb 函数，将 HTTP 请求包转化为 AJP 包。
4. 将当前 ajp_worker 的状态更新。
5. 调用 ajp_send_request 函数。
6. 如果发送成功，调用 ajp_get_reply 函数，等待并接受 AJP 包。
7. 如果成功，更新当前状态。

3.4.2 ajp_worker 的 ajp_send_request 函数

它的处理过程如下：

1. 调用 jk_is_socket_connected 函数，检查即将发送的 socket 是否有效。它的检查过程是运用 select 函数，等 1 微妙，如果成功返回说明，该 socket 的文件描述符至少

在内核中还有效。

2. 如果上一次发送超时，则调用 `ajp_handle_cping_cpong` 函数来判断后端连接是否有效。该函数主要是发出了一个 `AJP13_CPING_REQUEST` 类型的 AJP 包，如果 tomcat 收到，应返回一个 `AJP13_CPONG_REPLY` 的包。
3. 如果上述的测试都成功，调用 `ajp_connection_tcp_send_message`，发送 AJP 包。

3.4.3 ajp_worker 的 ajp_get_reply 函数

它的处理过程如下：

1. 调用 `jk_is_input_event`，用 `select` 等待接受数据，直到 socket 有接受数据。
2. 调用 `ajp_connection_tcp_get_message`，接受 AJP 包，并检查协议头。
3. 调用 `ajp_process_callback`，对该 AJP 包进行解析处理。其处理过程如下：
 - i. 如果是 `JK_AJP13_SEND_HEADERS` 包，将其解包成 HTTP 包头。如果没有错误，就调用 `jk_ws_service->start_response()` 函数发送 HTTP 回复的 head。返回 `JK_AJP13_SEND_HEADERS`。
 - ii. 如果是 `JK_AJP13_SEND_BODY_CHUNK` 包，调用 `jk_ws_service->write` 发送 HTTP 回复的 body。返回 `JK_AJP13_NO_RESPONSE`。
 - iii. 如果是 `JK_AJP13_GET_BODY_CHUNK` 包，说明还有数据要发送到 tomcat，调用 `ajp_read_into_msg_buff`，继续发送数据。返回 `JK_AJP13_HAS_RESPONSE`。
 - iv. 如果是 `JK_AJP13_END_RESPONSE` 包，就说明 tomcat 的回复已经完成。返回 `JK_AJP13_END_RESPONSE`。

3.5 jk_ws_service 的一些处理函数

3.5.1 jk_ws_service 的 ws_start_response 函数

该函数主要是构建 HTTP 回复包的头部。

3.5.2 jk_ws_service 的 ws_write 函数

调用 apache 的 `ap_rwrite` 函数，发送整个 HTTP 包。

3.5.3 jk_ws_service 的 ws_read 函数

调用 apache 的 `ap_get_client_block` 读入全部的 request body 数据。

4 mod_jk 的主要数据结构

4.1 lb_worker

`lb_worker` 是负载均衡 worker。主要负责调配其名下的一些 `lb_sub_worker`。

```
dist/common/jk_lb_worker.h
```

```
struct lb_worker
{
    /* jk 模块通用 worker 结构体。包含一些回调函数，不同的 worker 有不同的函数实现 */
    jk_worker_t worker;
    /* Shared memory worker data */
    jk_shm_lb_worker_t *s;

    char          name[JK_SHM_STR_SIZ+1];
    /* Sequence counter starting at 0 and increasing
     * every time we change the config
     */
    volatile unsigned int sequence;

    jk_pool_t p;
    jk_pool_atom_t buf[TINY_POOL_SIZE];

    JK_CRIT_SEC cs;
    /*其名下的 workers*/
    lb_sub_worker_t *lb_workers;
    unsigned int num_of_workers;
    int          sticky_session;
    int          sticky_session_force;
    int          recover_wait_time;
    int          max_reply_timeouts;
    int          retries;
    int          retry_interval;
    int          lbmethod;
    int          lblock;
    int          maintain_time;
    unsigned int max_packet_size;
    unsigned int next_offset;
    /* Session cookie */
    char          session_cookie[JK_SHM_STR_SIZ+1];
    /* Session path */
    char          session_path[JK_SHM_STR_SIZ+1];
};
```

lb_sub_worker 是由 lb_worker 支配的 ajp_worker。当然 lb_sub_worker 有不同的类型。它们的实际类型存储在 ajp_worker 中。

```
dist/common/jk_lb_worker.h
```

```
struct lb_sub_worker
{
    /* 包含 ajp_worker 的回调函数及其 ajp_worker 结构体*/
};
```

```

jk_worker_t *worker;
/* Shared memory worker data */
jk_shm_lb_sub_worker_t *s;

char          name[JK_SHM_STR_SIZ+1];
/* Sequence counter starting at 0 and increasing
 * every time we change the config
 */
volatile unsigned int sequence;

/* route */
char  route[JK_SHM_STR_SIZ+1];
/* worker domain */
char  domain[JK_SHM_STR_SIZ+1];
/* worker redirect route */
char  redirect[JK_SHM_STR_SIZ+1];
/* worker distance */
int distance;
/* current activation state (config) of the worker */
int activation;
/* Current lb factor */
int lb_factor;
/* Current worker index */
int i;
/* Current lb reciprocal factor */
jk_uint64_t lb_mult;
};
typedef struct lb_sub_worker lb_sub_worker_t;

```

4.2 *ajp_worker*

ajp_worker 是具体工作的 worker，对应后端一个 tomcat 应用服务。

dist/common/jk_ajp_common.c

```

struct ajp_worker
{
    /* 包含 ajp_worker 的回调函数*/
    jk_worker_t worker;
    /* Shared memory worker data */
    jk_shm_ajp_worker_t *s;

    char          name[JK_SHM_STR_SIZ+1];
    /* Sequence counter starting at 0 and increasing
     * every time we change the config
     */
    volatile unsigned int sequence;

```

```

jk_pool_t p;
jk_pool_atom_t buf[TINY_POOL_SIZE];

JK_CRIT_SEC cs;

struct sockaddr_in worker_inet_addr;    /* Contains host and port */
unsigned connect_retry_attempts;
const char *host;
int port;
int maintain_time;
/*
 * Open connections cache...
 *
 * 1. Critical section object to protect the cache.
 * 2. Cache size.
 * 3. An array of "open" endpoints.
 */
unsigned int ep_cache_sz;
unsigned int ep_mincache_sz;
unsigned int ep_maxcache_sz;
int cache_acquire_timeout;
ajp_endpoint_t **ep_cache;

int proto;          /* PROTOCOL USED AJP13/AJP14 */

jk_login_service_t *login;

/* Weak secret similar with ajp12, used in ajp13 */
const char *secret;

/*
 * Post physical connect handler.
 * AJP14 will set here its login handler
 */
int (*logon) (ajp_endpoint_t * ae, jk_logger_t *l);

/*
 * Handle Socket Timeouts
 */
int socket_timeout;
int socket_connect_timeout;
int keepalive;
int socket_buf;
/*
 * Handle Cache Timeouts
 */

```

```

int cache_timeout;

/*
 * Handle Connection/Reply Timeouts
 */
int connect_timeout; /* connect cping/cpong delay in ms (0 means
disabled) */
int reply_timeout; /* reply timeout delay in ms (0 means disabled) */
int prepost_timeout; /* before sending a request cping/cpong timeout
delay in ms (0 means disabled) */
int conn_ping_interval; /* interval for sending keepalive cping packets on
 * unused connection */

int ping_timeout; /* generic cping/cpong timeout. Used for keepalive
packets or
 * as default for boolean valued connect and
prepost timeouts.
 */
unsigned int ping_mode; /* Ping mode flags (which types of cpings should
be used) */
/*
 * Recovery options
 */
unsigned int recovery_opts;

/*
 * Public property to enable the number of retry attempts
 * on this worker.
 */
int retries;

unsigned int max_packet_size; /* Maximum AJP Packet size */

int retry_interval; /* Number of milliseconds to sleep before
doing a retry */

/*
 * HTTP status that will cause failover (0 means disabled)
 */
unsigned int http_status_fail_num;
int http_status_fail[JK_MAX_HTTP_STATUS_FAILS];
};

```

4.3 status_worker

status_worker 用于产生一些状态的统计信息。
dist/common/jk_statuc.c

```

struct status_worker
{
    jk_pool_t          p;
    jk_pool_atom_t    buf[TINY_POOL_SIZE];
    const char        *name;
    const char        *css;
    const char        *ns;
    const char        *xmlns;
    const char        *doctype;
    const char        *prefix;
    int               read_only;
    char              **user_names;
    unsigned int      num_of_users;
    int               user_case_insensitive;
    jk_uint32_t       good_mask;
    jk_uint32_t       bad_mask;
    jk_worker_t       worker;
    jk_worker_env_t   *we;
};

```

4.4 *jk_worker* 和 *jk_endpoint*

jk_worker 是所有 worker 通用结构体名称，主要包含的是一些函数指针。它是一个类似 java 中抽象类的概念，各成员函数在从 *factory* 函数生产时初始化。

dist/common/jk_service.h

```

struct jk_worker
{
    /*
     * A 'this' pointer which is used by the subclasses of this class to
     * point to data/functions which are specific to a given protocol
     * (e.g. ajp12 or ajp13 or ajp14).
     */
    void *worker_private; /* 指向 ajp_worker, lb_worker 结构体 ... */

    int type;
    /*
     * For all of the below (except destroy), the first argument is
     * essentially a 'this' pointer.
     */

    /* 先于 init 函数调用，用于分配各类 worker 结构体的内存，作必要的初始化
     * Given a worker which is in the process of being created, and a list
     * of configuration options (or 'properties'), check to see if it the
     * options are. This will always be called before the init() method.
     * The init/validate distinction is a bit hazy to me.
     */

```

```

    * See jk_ajp13_worker.c/jk_ajp14_worker.c and jk_worker.c-
>wc_create_worker()
    */
    int (JK_METHOD * validate) (jk_worker_t *w,
                                jk_map_t *props,
                                jk_worker_env_t *we, jk_logger_t *l);

    /*
    * Update worker either from jk_status or reloading from workers.properties
    */
    int (JK_METHOD * update) (jk_worker_t *w,
                                jk_map_t *props,
                                jk_worker_env_t *we, jk_logger_t *l);

    /*
    * Do whatever initialization needs to be done to start this worker up.
    * Configuration options are passed in via the props parameter.
    */
    int (JK_METHOD * init) (jk_worker_t *w,
                            jk_map_t *props,
                            jk_worker_env_t *we, jk_logger_t *l);

    /*
    * Obtain an endpoint to service a particular request.  A pointer to
    * the endpoint is stored in pend.
    */
    int (JK_METHOD * get_endpoint) (jk_worker_t *w,
                                    jk_endpoint_t **pend, jk_logger_t *l);

    /*
    * Shutdown this worker.  The first argument is not a 'this' pointer,
    * but rather a pointer to 'this', so that the object can be free'd (I
    * think -- though that doesn't seem to be happening.  Hmmm).
    */
    int (JK_METHOD * destroy) (jk_worker_t **w, jk_logger_t *l);

    /*
    * Maintain this worker.
    */
    int (JK_METHOD * maintain) (jk_worker_t *w, time_t now, jk_logger_t *l);
};

```

/* jk_endpoint 可以称之为通用终端服务结构体，各类 worker 可能有自己的 endpoint 结构体。比如 ajp_worker，该终端是用来做具体工作的，比如发出请求，接收 AJP 数据等等。如

```

果是 lb_worker，该终端的作用是找出合适 ajp_worker，把任务交给它。*/
struct jk_endpoint
{
    jk_uint64_t rd;
    jk_uint64_t wr;

    /*
     * Flag to pass back recoverability status from
     * a load balancer member to the load balancer itself.
     * Depending on the configuration and request status
     * recovery is not allowed.
     */
    int recoverable;

    /*
     * A 'this' pointer which is used by the subclasses of this class to
     * point to data/functions which are specific to a given protocol
     * (e.g. ajp12 or ajp13 or ajp14).
     */
    void *endpoint_private;

    /* 该函数是具体的服务函数
     * Forward a request to the servlet engine. The request is described
     * by the jk_ws_service_t object.
     * is_error is either 0 meaning recoverable or set to
     * the HTTP error code.
     */
    int (JK_METHOD * service) (jk_endpoint_t *e,
                               jk_ws_service_t *s,
                               jk_logger_t *l, int *is_error);

    /*
     * Called when this particular endpoint has finished processing a
     * request. For some protocols (e.g. ajp12), this frees the memory
     * associated with the endpoint. For others (e.g. ajp13/ajp14), this can
     * return the endpoint to a cache of already opened endpoints.
     *
     * Note that the first argument is not a 'this' pointer, but is
     * rather a pointer to a 'this' pointer. This is necessary, because
     * we may need to free this object.
     */
    int (JK_METHOD * done) (jk_endpoint_t **p, jk_logger_t *l);
};

```

4.5 *jk_ws_service*

该结构体是与 apache 相关的一些参数或函数，面向 HTTP 协议，与 AJP 协议不太相关。

dist/common/jk_service.h

```
struct jk_ws_service
{
    /*
     * A 'this' pointer which is used by the subclasses of this class to
     * point to data which is specific to a given web server platform
     * (e.g. Apache or IIS).
     */
    void *ws_private;

    /*
     * Provides memory management. All data specific to this request is
     * allocated within this pool, which can then be reclaimed at the end
     * of the request handling cycle.
     *
     * Alive as long as the request is alive.
     */
    jk_pool_t *pool;

    /*
     * CGI Environment needed by servlets
     */
    const char *method;
    const char *protocol;
    char *req_uri;
    const char *remote_addr;
    const char *remote_host;
    const char *remote_user;
    const char *auth_type;
    const char *query_string;
    const char *server_name;
    unsigned server_port;
    char *server_software;
    jk_uint64_t content_length; /* 64 bit integer that represents the content */
    /* length should be 0 if unknown. */
    unsigned is_chunked; /* 1 if content length is unknown (chunked rq) */
    unsigned no_more_chunks; /* 1 if last chunk has been read */
    jk_uint64_t content_read; /* number of bytes read */

    /*
     * SSL information
     */
}
```

```

* is_ssl      - True if request is in ssl connection
* ssl_cert    - If available, base64 ASN.1 encoded client certificates.
* ssl_cert_len - Length of ssl_cert, 0 if certificates are not available.
* ssl_cipher  - The ssl cipher suite in use.
* ssl_session - The ssl session string
*
* In some servers it is impossible to extract all this information, in this
* case, we are passing NULL.
*/
int is_ssl;
char *ssl_cert;
unsigned ssl_cert_len;
char *ssl_cipher;
char *ssl_session;

/*
 * SSL extra information for Servlet 2.3 API
 *
 * ssl_key_size - ssl key size in use
 */
int ssl_key_size;

/*
 * Headers, names and values.
 */
char **headers_names; /* Names of the request headers */
char **headers_values; /* Values of the request headers */
unsigned num_headers; /* Number of request headers */

/*
 * Request attributes.
 *
 * These attributes that were extracted from the web server and are
 * sent to Tomcat.
 *
 * The developer should be able to read them from the ServletRequest
 * attributes. Tomcat is required to append org.apache.tomcat. to
 * these attribute names.
 */
char **attributes_names; /* Names of the request attributes */
char **attributes_values; /* Values of the request attributes */
unsigned num_attributes; /* Number of request attributes */

/*
 * The route is in use when the adapter load balance among

```

```

    * several workers. It is the ID of a specific target in the load balance
    * group. We are using this variable to implement target session
    * affinity
    */
const char *route;

/* Temp solution for auth. For native1 it'll be sent on each request,
   if an option is present. For native2 it'll be sent with the first
   request. On java side, both cases will work. For tomcat3.2 or
   a version that doesn't support secret - don't set the secret,
   and it'll work.
   */
const char *secret;

/*
   * Area to get POST data for fail-over recovery in POST
   */
jk_msg_buf_t *reco_buf;
int reco_status;

/*
   * If set call flush after each write
   */
int flush_packets;

/*
   * If set call flush after AJP13_SEND_HEADERS.
   */
int flush_header;

/*
   * service extensions
   */
svc_extension_t extension;

/*
   * JK_TRUE if response headers have been sent back
   */
int response_started;

/*
   * JK_TRUE if response should not be send to the client
   */
int response_blocked;

/*
```

```

    * HTTP status sent from container.
    */
int http_response_status;

/* Uri worker map. Added for virtual host support
   */
jk_uri_worker_map_t *uw_map;

/* 下面这些回调函数实现都可以在 mod_jk.c 找到
   * Callbacks into the web server. For each, the first argument is
   * essentially a 'this' pointer. All return JK_TRUE on success
   * and JK_FALSE on failure.
   */
/*
   * Send the response headers to the browser.
   */
int (JK_METHOD * start_response) (jk_ws_service_t *s,
                                   int status,
                                   const char *reason,
                                   const char *const *header_names,
                                   const char *const *header_values,
                                   unsigned num_of_headers);

/*
   * Read a chunk of the request body into a buffer. Attempt to read len
   * bytes into the buffer. Write the number of bytes actually read into
   * actually_read.
   */
int (JK_METHOD * read) (jk_ws_service_t *s,
                        void *buffer,
                        unsigned len, unsigned *actually_read);

/*
   * Write a chunk of response data back to the browser.
   */
int (JK_METHOD * write) (jk_ws_service_t *s,
                          const void *buffer, unsigned len);

/*
   * Flush a chunk of response data back to the browser.
   */
void (JK_METHOD * flush) (jk_ws_service_t *s);

/*
   * Done with sending response back to the browser.
   */

```

```

void (JK_METHOD * done) (jk_ws_service_t *s);

/*
 * If set do not reuse socket after each full response
 */
int disable_reuse;

/*
 * Add more data to log facilities.
 */
void (JK_METHOD * add_log_items) (jk_ws_service_t *s,
                                   const char *const *log_names,
                                   const char *const *log_values,
                                   unsigned num_of_items);

/*
 * Iterate through all vhosts
 */
void *(JK_METHOD * next_vhost) (void *d);

/*
 * String representation of a vhost
 */
void (JK_METHOD * vhost_to_text) (void *d, char *buf, size_t len);

/*
 * Get uw_map associated with a vhost
 */
jk_uri_worker_map_t *(JK_METHOD * vhost_to_uw_map) (void *d);
};

```

4.6 共享内存中一些数据结构

共享内存中的数据结构基本上都是记录之用。有些参数在运行中会实时变化。

dist/common/jk_shm.h

```

/** jk shm generic worker record structure */
struct jk_shm_worker_header
{
    int    id;
    int    type;
    /* worker name */
    char   name[JK_SHM_STR_SIZ+1];
    /* Sequence counter starting at 0 and increasing
     * every time we change the config
     */
    volatile unsigned int sequence;
};

```

```

typedef struct jk_shm_worker_header jk_shm_worker_header_t;

/** jk shm ajp13/ajp14 worker record structure */
struct jk_shm_ajp_worker
{
    jk_shm_worker_header_t h;

    /* Configuration data mirrored from ajp_worker */
    int cache_timeout;
    int connect_timeout;
    int reply_timeout;
    int prepost_timeout;
    unsigned int recovery_opts;
    int retries;
    int retry_interval;
    unsigned int max_packet_size;
    /* current error state (runtime) of the worker */
    volatile int state;
    /* Statistical data */
    /* Number of currently busy channels */
    volatile int busy;
    /* Maximum number of busy channels
       该参数并非我们的maxbusy，它是该worker自程序运行以来busy的最大值 */
    volatile int max_busy;
    volatile time_t error_time;
    /* Number of bytes read from remote */
    volatile jk_uint64_t readed;
    /* Number of bytes transferred to remote */
    volatile jk_uint64_t transferred;
    /* Number of times the worker was used */
    volatile jk_uint64_t used;
    /* Number of times the worker was used - snapshot during maintenance */
    volatile jk_uint64_t used_snapshot;
    /* Number of non 200 responses */
    volatile jk_uint32_t errors;
    /* Decayed number of reply_timeout errors */
    volatile jk_uint32_t reply_timeouts;
    /* Number of client errors */
    volatile jk_uint32_t client_errors;
    /* Last reset time */
    volatile time_t last_reset;
    volatile time_t last_maintain_time;
};
typedef struct jk_shm_ajp_worker jk_shm_ajp_worker_t;

/** jk shm lb sub worker record structure */

```

```

struct jk_shm_lb_sub_worker
{
    jk_shm_worker_header_t h;

    /* route */
    char    route[JK_SHM_STR_SIZ+1];
    /* worker domain */
    char    domain[JK_SHM_STR_SIZ+1];
    /* worker redirect route */
    char    redirect[JK_SHM_STR_SIZ+1];
    /* Number of currently busy channels */
    volatile int busy;
    /* worker distance */
    volatile int distance;
    /* current activation state (config) of the worker */
    volatile int activation;
    /* current error state (runtime) of the worker */
    volatile int state;
    /* Current lb factor */
    volatile int lb_factor;
    /* 我们的参数加在这里*/
    volatile int maxbusy;
    /* Current lb reciprocal factor */
    volatile jk_uint64_t lb_mult;
    /* Current lb value */
    volatile jk_uint64_t lb_value;
    /* Statistical data */
    volatile time_t error_time;
    /* Number of times the worker was elected - snapshot during maintenance */
    volatile jk_uint64_t  elected_snapshot;
    /* Number of non 200 responses */
    volatile jk_uint32_t  errors;
};

typedef struct jk_shm_lb_sub_worker jk_shm_lb_sub_worker_t;

/** jk shm lb worker record structure */
struct jk_shm_lb_worker
{
    jk_shm_worker_header_t h;

    /* Number of currently busy channels, 该值是其名下 ajp_worker 的 busy 值之和*/
    volatile int busy;
    /* Maximum number of busy channels, 该值是其名下 ajp_worker 的 max_busy 值之和
*/
    volatile int max_busy;
    int    sticky_session;

```

```
int    sticky_session_force;
int    recover_wait_time;
int    max_reply_timeouts;
int    retries;
int    retry_interval;
int    lbmethod;
int    lblock;
unsigned int max_packet_size;
/* Last reset time */
volatile time_t last_reset;
volatile time_t last_maintain_time;
/* Session cookie */
char    session_cookie[JK_SHM_STR_SIZ+1];
/* Session path */
char    session_path[JK_SHM_STR_SIZ+1];
};
typedef struct jk_shm_lb_worker jk_shm_lb_worker_t;
```

参考链接:

1. <http://tomcat.apache.org/tomcat-5.5-doc/apr.html>
Tomcat 的 native 库由 C 语言开发，使用了 APR 库和 openssl 库，以动态链接库的形式由 java 调用。
2. <http://tomcat.apache.org/connectors-doc/ajp/ajpv13a.html>
3. <http://tomcat.apache.org/connectors-doc/reference/workers.html>

修改日志:

初次发布: 2009-3-11

后记:

由于初次接触 mod_jk，对 java 也知之甚少，错误在所难免，如有错误，请给我发邮件：
yaoweibin@gmail.com

