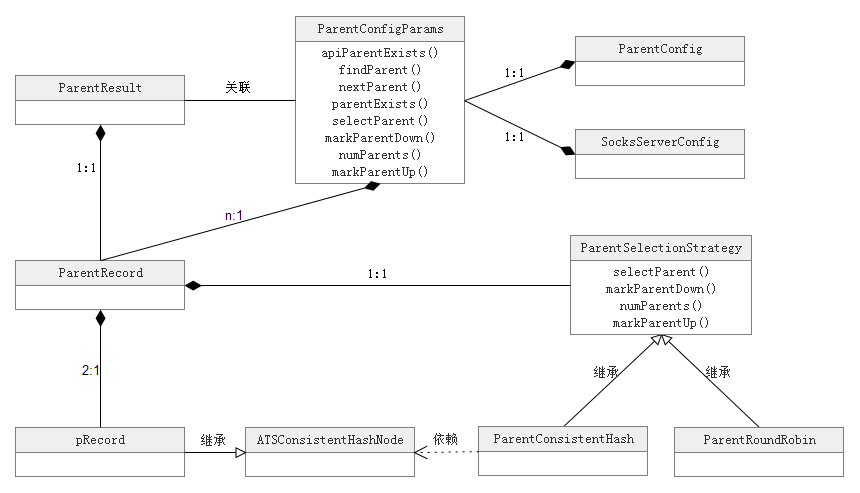
# parent父代理源码学习

## 文档编写者：曹泽敏

**基于ATS7.0.0代码研究(与ATS6.1.1相差不大)**

## UML图:



## 父代理的相关类介绍:

ParentSelectionStrategy 类, 基础工厂类

虚函数成员方法：

virtual void selectParent(const ParentSelectionPolicy \*policy, bool firstCall, ParentResult \*result, RequestData \*rdata) = 0;

virtual void markParentDown(const ParentSelectionPolicy \*policy, ParentResult \*result) = 0;

virtual uint32\_t numParents(ParentResult \*result) const = 0;

virtual void markParentUp(ParentResult \*result) = 0;

ParentConsistentHash 类, 内容哈希模块

相关类介绍：

ATSConsistentHash 类： （一致性内容哈希 和 其哈希配置存储）

将name域名结点node通过weight和自增变量值,计算出很多个离散的hash值出来,并存储到map中；

name（域名）查找时,先计算其哈希值,再通过map的 lower\_bound 函数找出对应的node结点。

成员变量:

ATSHash64Sip24 hash[2]; //哈希算法的指定, 对应 parent 和 secondary\_parent

ATSConsistentHash \*chash[2];

//配置内容哈希值存储指针, 对应 parent 和 secondary\_parent, 对象实例化在 ParentConsistentHash::ParentConsistentHash() 中

pRecord \*parents[2];

//存储 ParentRecord 的 parents 和 secondary\_parents 指针, 值赋值可参考 ParentConsistentHash::ParentConsistentHash()

bool foundParents[2][MAX\_PARENTS]; //此值暂时未使用

bool ignore\_query; //请求URL哈希时是否忽略URL的参数部分, 值赋值可参考 ParentConsistentHash::ParentConsistentHash()

static const int PRIMARY = 0; //常变量

static const int SECONDARY = 1; //常变量

内容HASH分为 PRIMARY(主) 和 SECONDARY(备) 两种选择

成员方法:

uint64\_t getPathHash(HttpRequestData \*hrdata, ATSHash64 \*h); //通过请求 hrdata 获取次请求域名对应的哈希值

void selectParent(const ParentSelectionPolicy \*policy, bool firstCall, ParentResult \*result, RequestData \*rdata);

//用请求 rdata 通过哈希的方式在 parents[2] 查找出合适的父IP(或域名)

void markParentDown(const ParentSelectionPolicy \*policy, ParentResult \*result);

//通过 policy 和 result 中的信息,

//来更新 result 对应 ParentRecord 的 parents 和 secondary\_parents 中父IP(或域名)的不可用状态

uint32\_t numParents(ParentResult \*result) const;

//返回正在查询的 parents[PRIMARY/SECONDARY] 在 parent.config　中配置父代理IP或域名的个数

void markParentUp(ParentResult \*result);

//通过 result 中的信息, 来更新 result 对应 ParentRecord 的 parents 和 secondary\_parents 中父IP(或域名)的可用状态

ParentRoundRobin 类, 轮询模块

成员变量:

ParentRR\_t round\_robin\_type;

//枚举变量, 对应 parent.config 中的 round\_robin 参数, 初始化可参考 ParentRecord::Init(), 值有: （具体示意参考 parent.config 官方文档）

// P\_NO\_ROUND\_ROBIN: 即 false, Round robin selection does not occur.

// P\_STRICT\_ROUND\_ROBIN: 即 strict, Traffic Server machines serve requests strictly in turn.

// P\_HASH\_ROUND\_ROBIN: 即 true, Traffic Server determines the parent based on client IP address.

// 上面三个对应 ParentRoundRobin

// P\_CONSISTENT\_HASH: 即 consistent\_hash, consistent hash of the url so that one parent is chosen for a given url. （对应 ParentConsistentHash ）

成员方法:

void selectParent(const ParentSelectionPolicy \*policy, bool firstCall, ParentResult \*result, RequestData \*rdata);

//通过请求 rdata 在 result->rec->parents 按指定方式查找可用的父代理信息, 并用 result 的其他变量存储和返回

void markParentDown(const ParentSelectionPolicy \*policy, ParentResult \*result);

//通过 policy 和 result 中的信息, 来更新 result->rec->parents 中父IP(或域名)的不可用状态

uint32\_t numParents(ParentResult \*result) const; //返回 result->rec->parents 在 parent.config　中配置父代理IP或域名的个数

void markParentUp(ParentResult \*result); //更新 result->rec->parents 中父IP(或域名)的可用状态

ParentConsistentHash 和 ParentRoundRobin 都继承于 ParentSelectionStrategy 类,使用的是 工厂方法 设计模式。

ATSConsistentHashNode 结构体, 结点, 就是个struct

bool available; //是否可用

char \*name; //域名指针

pRecord 结构体, 继承于 ATSConsistentHashNode ，主要是对 ATSConsistentHashNode 其他属性的扩展

存储 parent.config 中的 parent 或 secondary\_parent 的配置项

成员变量配置属性:

char hostname[MAXDNAME + 1]; //域名, 可以是IP

int port; //端口值

time\_t failedAt; //Mark the parent as down, 是原子操作

int failCount; //失败次数, 是原子操作

int32\_t upAt; //此值未使用

const char \*scheme; //A request URL protocol: http or https.

int idx; //表示 parent 或 secondary\_parent 的父代理的坐标索引

float weight; //表示 parent 或 secondary\_parent 的多个父代理的权重值, 只用于 consistent\_hash, 默认值为 1.0

ParentRecord 类, 一个对象对应一行 parent.config 配置

parent.config 的参数具体含义请参考官方文档

成员变量:

pRecord \*parents; //数组指针, 用于存 parent.config 的 parent 配置项

pRecord \*secondary\_parents; //数组指针, 用于存 parent.config 的 secondary\_parent 配置项, 只针对 consistent\_hash 生效

int num\_parents; //与 parents 对应, 保存 parent.config 的 parent 配置父代理的总数

int num\_secondary\_parents; //与 secondary\_parents 对应, 保存 parent.config 的 secondary\_parent 配置父代理的总数

const char \*scheme; //A request URL protocol: http or https. 用于存 parent.config 的 scheme 配置项

bool ignore\_query; //请求URL哈希时是否忽略URL的参数部分, 只针对 consistent\_hash 生效, 用于存 parent.config 的 qstring 配置项

volatile uint32\_t rr\_next; //强制轮询计数器, 只用于 ParentRoundRobin 的 P\_STRICT\_ROUND\_ROBIN 场景

bool go\_direct;

//对应 parent.config 的 go\_direct 参数, 默认值为 true, parent\_is\_proxy 为 cache servers 时才会生效

//true, 请求不走父直接回源站(PARENT\_DIRECT), requests bypass parent hierarchies and go directly to the origin server.

//false, 请求走父, requests do not bypass parent hierarchies.

bool parent\_is\_proxy; //对应 parent.config 的 parent\_is\_proxy 参数, 默认值为 true

//true 表示此行配置的是 cache servers

//false 表示此行配置的是 origin servers ,此时的 go\_direct 值是被忽略无效的

ParentSelectionStrategy \*selection\_strategy; //策略选择, 在 ParentConsistentHash 和 ParentRoundRobin 两种中选择

UnavailableServerResponseCodes \*unavailable\_server\_retry\_responses;

//对应 parent.config 的 unavailable\_server\_retry\_responses 参数,

//表示返回的状态码是否为不可用, 默认值为503, 配置值 statuscode(500< statuscode <600), 可以配多个值

ParentRetry\_t parent\_retry; //枚举类型, 对应 parent.config 的 parent\_retry 参数

//PARENT\_RETRY\_NONE: 初始值

//PARENT\_RETRY\_SIMPLE: 即 simple\_retry

//PARENT\_RETRY\_UNAVAILABLE\_SERVER: 即 unavailable\_server\_retry

//PARENT\_RETRY\_BOTH: 即 both, 为 simple\_retry 和 unavailable\_server\_retry 都选择

int max\_simple\_retries; //对应 parent.config 的 max\_simple\_retries 参数, 默认值为1, 与 parent\_retry 一起使用

int max\_unavailable\_server\_retries;

//对应 parent.config 的 max\_unavailable\_server\_retries 参数, 默认值为1, 与 parent\_retry 一起使用

成员方法:

config\_parse\_error Init(matcher\_line \*line\_info); //解析一行 parent.config 配置

bool DefaultInit(char \*val); //parent.config 默认配置的解析, ParentConfigParams 初始化时被 createDefaultParent() 调用

//默认值配置 records.config, [default\_var] proxy.config.http.parent\_proxies

void UpdateMatch(ParentResult \*result, RequestData \*rdata);

//根据 result->line\_number 和请求 rdata 来更新查找的 ParentRecord 配置

void Print(); //打印 ParentRecord 的配置信息, 主要打印 hostname, port, go\_direct, parent\_is\_proxy 参数

bool bypass\_ok() const; //返回 go\_direct 的值

const char \*ProcessParents(char \*val, bool isPrimary); //解析 parent.config 中的 parent 和 secondary\_parent 参数,

//并将值保存到 parents 和 secondary\_parents 中

ParentResult 类, 存储 ParentRecord 对象, 和一些其他的相关参数, 一个请求对应一个 ParentResult

ParentResult 与 ParentRecord 是多对一的关系

成员变量:

ParentResultType result; //枚举变量,表示请求查找父代理的类型,值为:

// PARENT\_UNDEFINED : 未指定父代理,初始化值

// PARENT\_DIRECT : 直接回源

// PARENT\_SPECIFIED : 指定了父代理包括api设置父代理

// PARENT\_AGENT : 此值暂时未使用

// PARENT\_FAIL : 指定父代理失败

const char \*hostname; //匹配配置中的主机名

int port; //匹配配置中的端口

bool retry; //是否进行重试

//private:

int line\_number; //rec 配置在 parent.config 中对应的行号

ParentRecord \*rec; //parent.config 中的一项配置

uint32\_t last\_parent; //最后一个父配置项

uint32\_t start\_parent; //第一个父配置项, 只在 ParentRoundRobin::selectParent() 中使用了

bool wrap\_around; //没有实际意义

// state for consistent hash.

int last\_lookup; //表示查找的是 PRIMARY 还是 SECONDARY, 只针对 consistent hash 生效

ATSConsistentHashIter chashIter[2];

//map的迭代器, 定义为:typedef std::map<uint64\_t, ATSConsistentHashNode \*>::iterator ATSConsistentHashIter;

//ATSConsistentHash 中 lookup\_by\_hashval 和 lookup 查找 ATSConsistentHashNode 时的迭代器, 不传此值也没问题

成员方法:

void reset(); //此 ParentResult 对象初始化

bool is\_api\_result() const; //判断是否调用了api设置父代理

bool is\_some() const; //判断 rec 是否为空, 并进行了特需校验

bool parent\_is\_proxy() const; //先判断 is\_api\_result(), 并返回是否为父代理

unsigned retry\_type() const; //先判断 is\_api\_result(), 并返回 parent\_retry 值

unsigned max\_retries(ParentRetry\_t method) const; //先判断 is\_api\_result(), 并通过 method 值返回最大重试次数

bool response\_is\_retryable(HTTPStatus response\_code) const; //通过 retry\_type() 和 状态码 判断源站是否可用

bool bypass\_ok() const; //先判断 is\_api\_result(), 再返回 go\_direct 的值

typedef ControlMatcher<ParentRecord, ParentResult> P\_table; //使用模板加载配置,配置模板可以参考 congestion 代码解析

ParentConfigParams 类，继承于 ConfigInfo, 用于配置管理,其对象的使用和删除是通过计数器来控制的

成员变量:

P\_table \*parent\_table;

//保存 parent.config 中的配置项，父配置文件 parent.config 指定参数为 proxy.config.http.parent\_proxy.file

ParentRecord \*DefaultParent; //父的默认配置, 可以通过 [default\_var] proxy.config.http.parent\_proxies 进行指定

//若 default\_var 参数未指定, 则无默认的父配置, 线上情况一般没有指定

ParentSelectionPolicy policy; //是对 records.config 中父配置的存储

//ParentRetryTime: [retry\_var] proxy.config.http.parent\_proxy.retry\_time ,

//ParentEnable: [enable\_var] proxy.config.http.parent\_proxy\_routing\_enable ,

//FailThreshold: [threshold\_var] proxy.config.http.parent\_proxy.fail\_threshold

成员方法:

bool apiParentExists(HttpRequestData \*rdata);

//判断此请求是否通过 rdata->api\_info (如: TSHttpTxnParentProxySet) 设置了父代理信息

void findParent(HttpRequestData \*rdata, ParentResult \*result);

//通过请求信息 rdata 在 parent\_table 中查找对应的父配置信息,

//调用 selectParent() 时 firstCall 为 true 即第一次查找, 主策略中查找, 并返回 result

void nextParent(HttpRequestData \*rdata, ParentResult \*result);

//通过请求信息 rdata 在对应的 result->rec 即 ParentRecord 中选择备用父策略

//不需要通过 rdata 在 parent\_table 中重新查找 ParentRecord ,直接使用 result->rec

//调用 selectParent() 时 firstCall 为 false 即第二次查找, 备用策略中查找, 并返回 result

bool parentExists(HttpRequestData \*rdata); //通过请求 rdata 调用 findParent() 查找是否有可用的父策略

void selectParent(bool firstCall, ParentResult \*result, RequestData \*rdata);

//根据 result 中 ParentRecord 配置信息调用 selection\_strategy->selectParent 去选择对应的父策略

//其实是对 ParentRecord 中的 selection\_strategy->selectParent 做了一次封装

void markParentDown(ParentResult \*result);

//根据 result 中 ParentRecord 配置信息调用 selection\_strategy->markParentDown 去标记指定父配置不可用

//其实是对 ParentRecord 中的 selection\_strategy->markParentDown 做了一次封装

uint32\_t numParents(ParentResult \*result);

//根据 result 中 ParentRecord 配置信息调用 selection\_strategy->numParents 返回可用父配置数

//其实是对 ParentRecord 中的 selection\_strategy->numParents 做了一次封装

void markParentUp(ParentResult \*result);

//根据 result 中 ParentRecord 配置信息调用 selection\_strategy->markParentUp 去标记指定父配置可用

//其实是对 ParentRecord 中的 selection\_strategy->markParentUp 做了一次封装

ParentConfig 类, 配置的读取类，包括热加载和配置内存管理和释放

成员变量:

static int m\_id; //用于在 configProcessor 中查找 ParentConfigParams 对象配置的标识

成员方法:

static void startup(); //配置读取入口, 热加载的回调函数注册, 调用 reconfigure() 读取 parent.config 配置

static void reconfigure(); //加载 parent.config 的配置项，并用 ParentConfigParams \*params 对象保存配置

//params 指向的内存使用配置处理器 configProcessor 保存并管理

static void print(); //打印所有的父配置项, 即 ParentConfigParams 对象里面的内容

static void set\_parent\_table(P\_table \*pTable, ParentRecord \*rec, int num\_elements); //只有声明没有实现也没有使用。

//测试了struct下面声明一个函数,没有实现且没有使用, gcc编译器是允许的。

static ParentConfigParams \*acquire(); //通过 m\_id 在 configProcessor 中获取 ParentConfigParams 对象配置

static void release(ParentConfigParams \*strategy); //通过计数器释放 ParentConfigParams 对象内存

SocksServerConfig 类： socks代理，与 ParentConfig 类相似,配置存储都是用的 ParentConfigParams 类,主要是用于网络层的 SocksProxy 代理模块。读取的配置文件为 socks.config, 参数指定[records.config]: proxy.config.socks.socks\_config\_file

## 配置读取和更新:

1. void ParentConfig::startup(); //parent.config 读取入口, 在 main() 函数中调用, 注册 records.config 参数热加载的回调函数

parent.config 的热加载可以参考 ConfigUpdateHandler 类, 具体如何热加载的待研究

2. void ParentConfig::reconfigure(); //配置读取

P\_table \*pTable = new P\_table(file\_var, modulePrefix, &http\_dest\_tags); //parent.config 配置的读取

params = new ParentConfigParams(pTable); //默认配置的加载

3. ControlMatcher<ParentRecord, ParentResult>::ControlMatcher(const char \*file\_var, const char \*name, const matcher\_tags \*tags, int flags\_in);

4. int ControlMatcher<ParentRecord, ParentResult>::BuildTable();

5. int ControlMatcher<ParentRecord, ParentResult>::BuildTableFromString(char \*str);

6. config\_parse\_error HostMatcher<ParentRecord, ParentResult>::NewEntry(matcher\_line \*line\_info);

7. config\_parse\_error ParentRecord::Init(matcher\_line \*line\_info);

## 父代理处理流程:

1. find\_server\_and\_update\_current\_info 函数先通过HTTP请求用 ParentConfigParams::findParent 找出对应的 parent 配置, 用 result 返回赋给 s->parent\_result

2. 重试时查找父配置用 ParentConfigParams::nextParent, 用 result 返回赋给 s->parent\_result

3. 标记此父配置不可用 ParentConfigParams::markParentDown 在 HttpTransact::handle\_response\_from\_parent 和 HttpTransact::PPDNSLookup 调用

4. 标记此父配置可用 ParentConfigParams::markParentUp 在 HttpTransact::handle\_response\_from\_parent 调用

5. ATS查询和修改父配置对外的接口类为 ParentConfigParams, 所有的父配置状态信息修改和重新选择父配置都是在 s->parent\_result 中操作