# https://www.yuque.com/kenliu-zhmwk/qc8gak/wk0z55

# 浅析 Dubbo 3.0 中对 URL 的优化

(从本地拷过来格式乱了一些，此处不调整了，后续文章改动后会放到官网上)

## 回顾 Dubbo 2.7

在解释 Dubbo 3.0 对 URL 进行的优化之前，让我们来先回顾一下 Dubbo 2.7 中 URL 的实现。

### URL 的实现

了解过的人可能知道，在 2.7 中，URL 只有一个类，存储了所有相关的信息，不过我们今天的主角是下面几个属性。

public class URL implements Serializable {

protected String protocol;

protected String username;

protected String password;

protected String host;

protected int port;

protected String path;

private final Map<String, String> parameters;

// 省略其他无关属性

}

    那么这种结构存在了什么问题呢？Dubbo 3.0 又为什么需要对它进行优化呢？

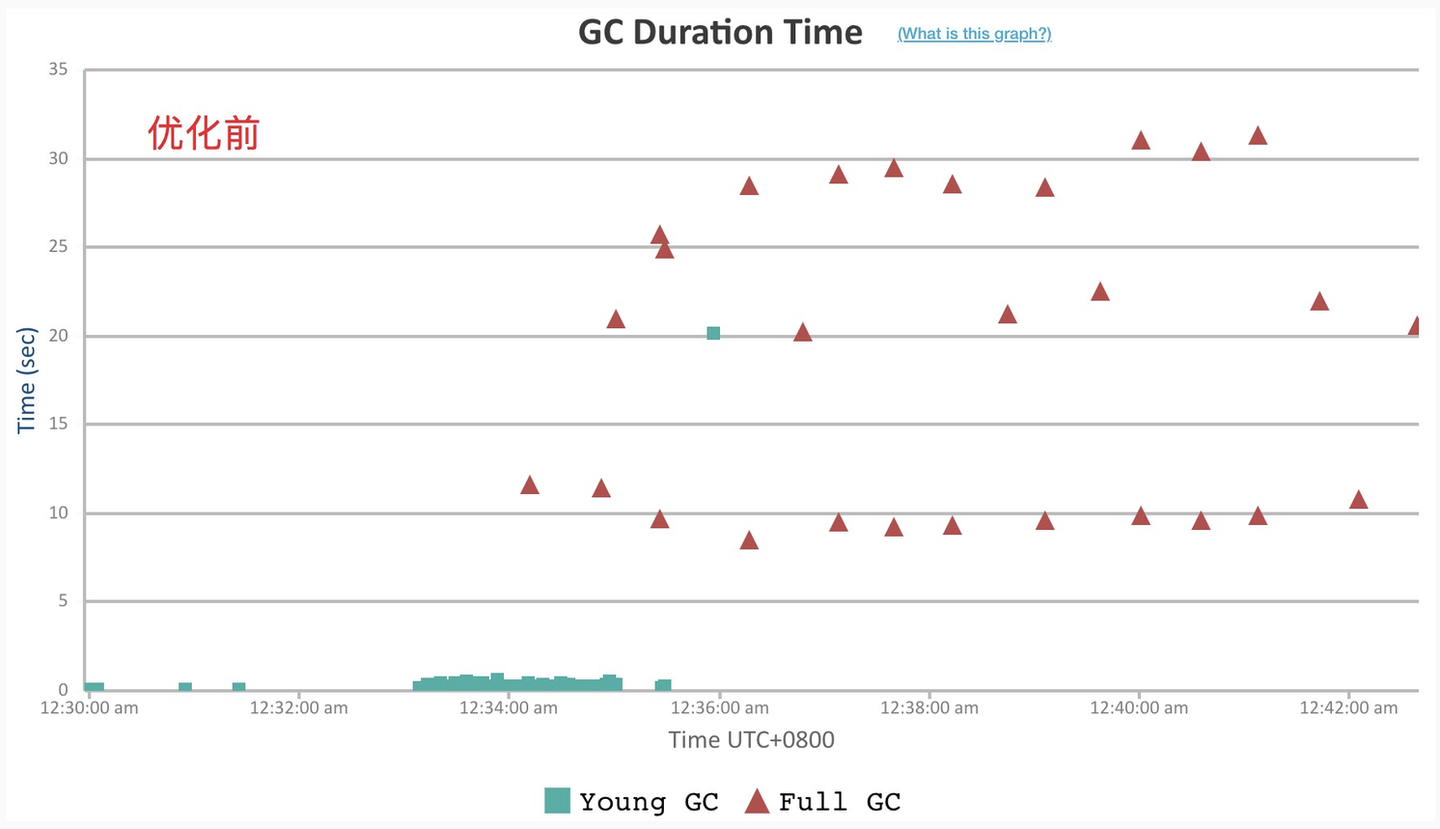
### 存在的问题

其实在 Dubbo 2.7 中 URL 的结构并不存在任何问题，主要是针对某些特定场景可能会出现一些性能问题。在 Dubbo 2.7 中，Provider 是接口级的，并且任一 Provider 实例接口参数的变更都会导致 Consumer 重新获取所有 Provider 的信息并且全部重新生成一遍 URL，这明显是不合适的。

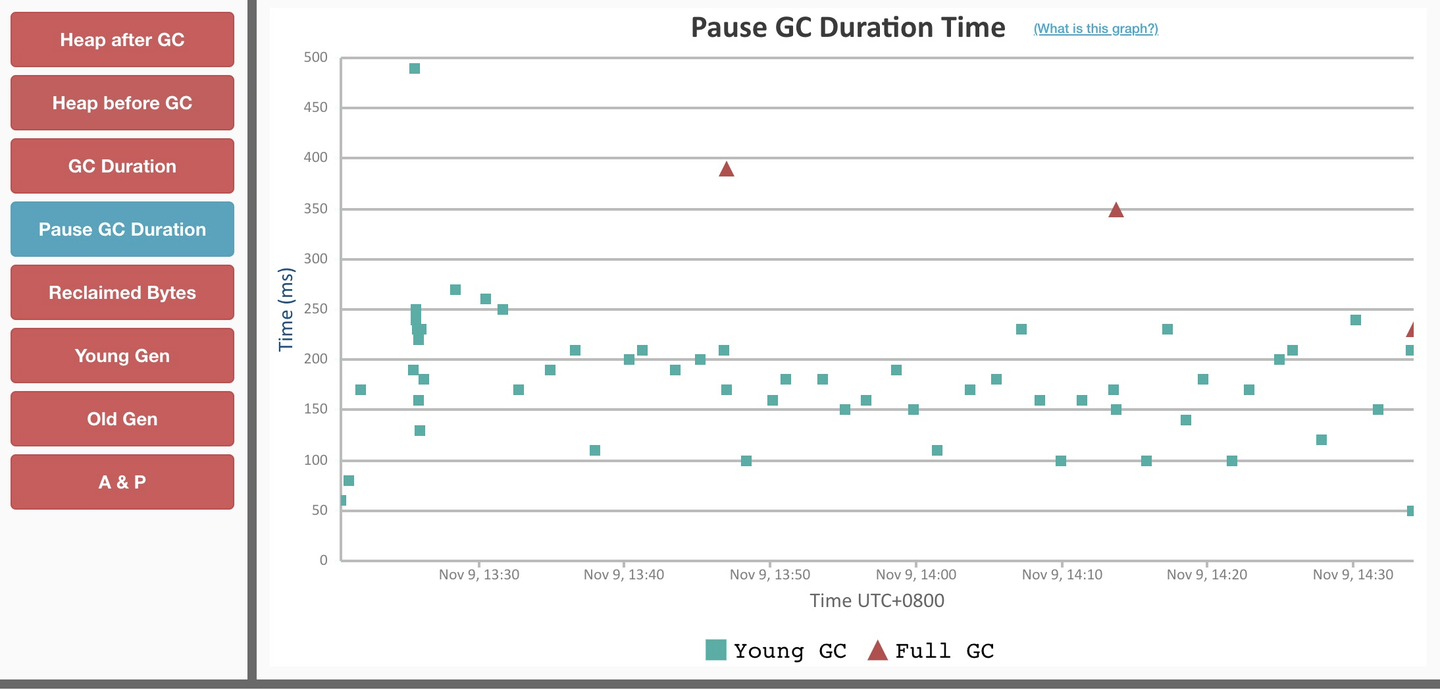
那么，请思考以下两种场景：

1. 某个 Consumer 依赖大量的 Provider，并且其中某个 Provider 因为网络等原因频繁上下线。
2. 当频繁的扩容缩容导致的 Provider 频繁变更时。

上面任一场景都会导致大批量的 URL 一直不断的创建，且有很大一部分原来便存在，导致了不必要的内存及运行时间的损耗。在这里我引用[《Dubbo 3.0 前瞻：服务发现支持百万集群，带来可伸缩微服务架构》](https://zhuanlan.zhihu.com/p/345626851" \t "/Users/gy821075/Documents\\x/_blank)这篇文章中的一副图来说明，该图模拟了在220万个 Provider 接口的情况下，接口数据不断变更导致的 Consumer 端的消耗，我们看到整个 Consumer 端几乎被 Full GC 占满了，严重影响了性能。



那么我们再来看看 Dubbo 3.0 中对 URL 进行优化后同一个环境下的压测结果，如下图所示。

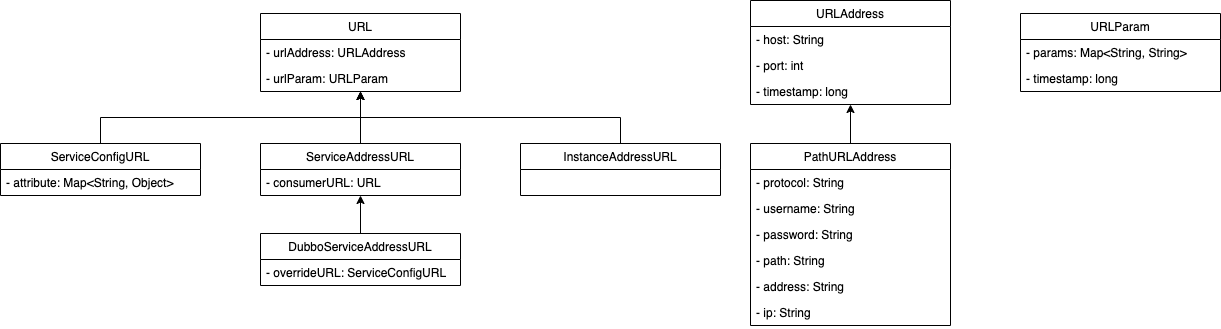


我们明显可以看到 Full GC 的频率减少到了只有 3 次，大大提升了性能。当然，该文章中还有其他方面的对比，此处便不一一引用了，感兴趣的读者可以自行去阅读该文章。那么接下来我们就来看看 Dubbo 3.0 具体做了哪些优化吧。

## Dubbo 3.0 中的URL

在讲解具体优化策略之前，我们先来看一看 URL 的最新实现（此处只标出了几个重点字段，不包含全部）。

### URL 的实现



我们可以看到，在 Dubbo 2.7 的 URL 中的几个重要属性在 Dubbo 3.0 中已经不存在了，取而代之的是 URLAddress 和 URLParam 两个类。原来的 parameters 属性被移动到了 URLParam 中的 params，其他的属性则移动到了 URLAddress 及其子类中。

再来介绍 URL 新增的 3 个子类，其中 InstanceAddressURL 属于应用级接口地址，本篇章中不做介绍。

而 ServiceConfigURL 及 ServiceAddressURL 主要的差别就是，ServiceConfigURL 是程序读取配置文件时生成的 URL。而 ServiceAddressURL 则是注册中心推送一些信息（如 providers）过来时生成的 URL。

在这里我们顺便提一下为什么会有 DubboServiceAddressURL 这个子类，按照目前的结构来看，ServiceAddressURL 只有这一个子类，所以完全可以将他们两个的属性全都放到 ServiceAddressURL 中，那么为什么还要有这个子类呢？其实是 Dubbo 3.0 为了兼容 HSF 框架所设计的，抽象出了一个 ServiceAddressURL，而 HSF 框架则可以继承这个类，使用 HSFServiceAddressURL，当然，这个类目前没有体现出来，所以此处我们简单一提，不过多讲解。

那么，我们接下来就讨论一下 Dubbo 3.0 为什么要改为此种数据结构，并且做了什么优化吧！

### 进行的优化

#### 结构上的优化

我们看到原来的属性都被移到了 URLAddress 和 URLParam 里，但是 URL 的子类依然多了几个属性，这几个属性自然也是为了优化而新增的，那么这里就讲讲这几个属性的作用。

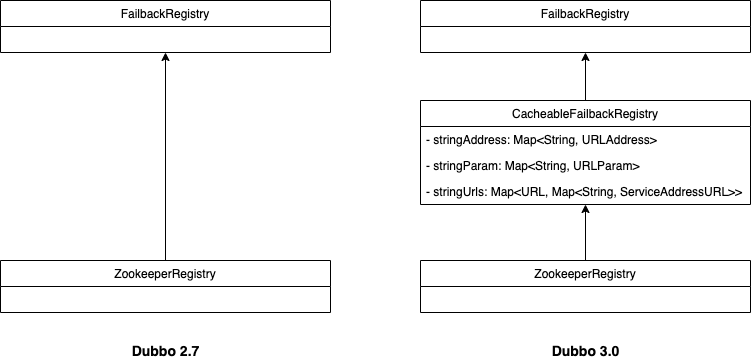
ServiceConfigURL：这个子类中新增了 attribute 这个属性，这个属性主要是针对 URLParam 的 params 做了冗余，仅仅只是将 value 的类型从 String 改为了 Object，减少了代码中每次获取 parameters 的格式转换消耗。

ServiceAddressURL：这个子类及其对应的其他子类中则新增了 overrideURL 和 consumerURL 属性。其中 consumerURL 是针对 consumer 端的配置信息，overrideURL 则是在 Dubbo Admin 上进行动态配置时写入的值，当我们调用 URL 的 getParameter() 方法时，优先级为 overrideURL > consumerURL > urlParam。在 Dubbo 2.7 时，动态配置属性会替换 URL 中的属性，及当你有大量 URL 时消耗也是不可忽视的，而此处的 overrideURL 则避免了这种消耗，因为所有 URL 都会共同使用同一个对象。

#### 多级缓存

缓存是 Dubbo 3.0 在 URL 上做的优化的重点，同时这部分也是解决上文中提出的两个场景导致的问题的地方，那么接下来我们就开始来介绍一下多级缓存的具体实现。

首先，多级缓存主要体现在 CacheableFailbackRegistry 这个类之中，它直接继承于 FailbackRegistry，以 Zookeeper 为例，我们看看 Dubbo 2.7 和 Dubbo 3.0 继承结构的区别。



可以看到在 CacheableFailbackRegistry 缓存中，我们新增了 3 个缓存属性 stringAddress，stringParam 和 stringUrls。我们来针对上文中的 2 个具体的场景来对这 3 个属性做具体的解析。

1、某个 Consumer 依赖大量的 Provider，并且其中某个 Provider 因为网络等原因频繁上下线：为了优化这个场景，我们主要是用了 stringUrls 这个属性，我们先来看看对应的代码片段。

// protected final Map<URL, Map<String, ServiceAddressURL>> stringUrls = new HashMap<>();

/\*\*

\* @param consumer consumer的URL对象

\* @param providers 注册中心推送的providers下的地址列表

\*/

protected List<URL> toUrlsWithoutEmpty(URL consumer, Collection<String> providers) {

// 先查出缓存中consumer目前对应的providers

Map<String, ServiceAddressURL> oldURLs = stringUrls.get(consumer);

Map<String, ServiceAddressURL> newURLs;

URL copyOfConsumer = removeParamsFromConsumer(consumer);

if (oldURLs == null) {

// 如果缓存中没有providers，则直接创建

newURLs = new HashMap<>();

for (String rawProvider : providers) {

rawProvider = stripOffVariableKeys(rawProvider);

// createURL方法中使用了stringAddress和stringParam缓存，后面讲解

ServiceAddressURL cachedURL = createURL(rawProvider, copyOfConsumer, getExtraParameters());

if (cachedURL == null) {

logger.warn("Invalid address, failed to parse into URL " + rawProvider);

continue;

}

newURLs.put(rawProvider, cachedURL);

}

} else {

newURLs = new HashMap<>((int)(oldURLs.size()/.75 + 1));

for (String rawProvider : providers) {

rawProvider = stripOffVariableKeys(rawProvider);

// 获取缓存中的provider

ServiceAddressURL cachedURL = oldURLs.remove(rawProvider);

if (cachedURL == null) {

// provider不存在则创建

// createURL方法中使用了stringAddress和stringParam缓存，后面讲解

cachedURL = createURL(rawProvider, copyOfConsumer, getExtraParameters());

if (cachedURL == null) {

logger.warn("Invalid address, failed to parse into URL " + rawProvider);

continue;

}

}

// provider存在则直接放入新的列表中

newURLs.put(rawProvider, cachedURL);

}

}

// 更新缓存

stringUrls.put(consumer, newURLs);

// 省略其他无关代码

}

我们看到当新的 providers 列表推送过来时，如果 URLParam 和 URLAddress 完全无变更的话，会直接省略 createURL() 步骤，从 stringUrls 中直接获取缓存的值，此处便能省略很多无用的 URL 创建过程，大大减少了 CPU 和内存的消耗。

2、当频繁的扩容缩容导致的 Provider 频繁变更时：当这种场景出现时，我们知道 URL 中的 URLAddress 属性肯定是不一样了，stringUrls 中便没有了缓存数据，那么我们就只能使用新的 URLAddress 和 URLParam 对象去创建 URL 对象了吗？不是的，因为在扩容所容的情况下，虽然 URLAddress 属性不同了，但是 URLParam 属性还是相同的，所以我们依然使用到了第二部分缓存。我们也来看看代码片段。

// protected final static Map<String, URLAddress> stringAddress = new ConcurrentHashMap<>();

// protected final static Map<String, URLParam> stringParam = new ConcurrentHashMap<>();

/\*\*

\* @param rawProvider provider的fullString

\* @param consumerURL consumerURL

\* @param extraParameters 额外的属性，此处忽略

\*/

protected ServiceAddressURL createURL(String rawProvider, URL consumerURL, Map<String, String> extraParameters) {

boolean encoded = true;

// 直接使用encode的URL来避免URLDecoder.decode的消耗（此处也是一个小优化，后面讲解）

int paramStartIdx = rawProvider.indexOf(ENCODED\_QUESTION\_MARK);

if (paramStartIdx == -1) {

// 如果ENCODED\_QUESTION\_MARK（即%3F）不存在，标记encoded为false

encoded = false;

}

String[] parts = URLStrParser.parseRawURLToArrays(rawProvider, paramStartIdx);

if (parts.length <= 1) {

logger.warn("Received url without any parameters " + rawProvider);

return DubboServiceAddressURL.valueOf(rawProvider, consumerURL);

}

String rawAddress = parts[0];

String rawParams = parts[1];

boolean isEncoded = encoded;

// 查找stringAddress缓存中是否有对应的值，没有则创建

URLAddress address = stringAddress.computeIfAbsent(rawAddress, k -> URLAddress.parse(k, getDefaultURLProtocol(), isEncoded));

address.setTimestamp(System.currentTimeMillis());

// 查找stringParam缓存中是否有对应的值，没有则创建

URLParam param = stringParam.computeIfAbsent(rawParams, k -> URLParam.parse(k, isEncoded, extraParameters));

param.setTimestamp(System.currentTimeMillis());

// 使用urlAddress和urlParam创建新的URL对象

ServiceAddressURL cachedURL = createServiceURL(address, param, consumerURL);

if (isMatch(consumerURL, cachedURL)) {

return cachedURL;

}

return null;

}

看代码可知，在这种场景下，创建新的 URL 对象时，我们依然复用了很多 URLParam 对象，只剩下了创建 URLAddress 的开销，而因为参数占用了 URL 中的一大部分空间，所以这种方式也能很有效的减少 CPU 和内存消耗。

#### 其他优化

除了上面提到的优化之外，其实另外还有两个小小的优化。

第一个是 URLStrParser 类中新增了 parseEncodedStr 和 parseEncodedParams 方法，这两个方法可以直接针对编码后的 URL 进行解析，减少了解码过程的消耗。

第二个则是 URL 变更后的通知机制增加了延迟，依然以 Zookeeper 为例，现在 ZookeeperRegistry 中的通知使用了 RegistryChildListenerImpl 方法来接受通知消息，在讲解具体的代码前，我们先来说一说 Zookeeper 的通知机制。当一个 Consumer Watcher 阻塞时，Zookeeper 的通知亦会阻塞，而当阻塞期间对应的节点（此处即为 providers）有多次变更时，Zookeeper 只会保留最后一次变更，在阻塞结束后通知该次变更。

接下来我们来看看它的实现。

public RegistryChildListenerImpl(URL consumerUrl, String path, NotifyListener listener, CountDownLatch latch) {

this.latch = latch;

notifier = new RegistryNotifier(ZookeeperRegistry.this.getDelay()) {

@Override

public void notify(Object rawAddresses) {

long delayTime = getDelayTime();

if (delayTime <= 0) {

this.doNotify(rawAddresses);

} else {

// 延迟一定时间后才去真正的通知更新

long interval = delayTime - (System.currentTimeMillis() - lastExecuteTime);

if (interval > 0) {

try {

Thread.sleep(interval);

} catch (InterruptedException e) {

// ignore

}

}

lastExecuteTime = System.currentTimeMillis();

this.doNotify(rawAddresses);

}

}

@Override

protected void doNotify(Object rawAddresses) {

ZookeeperRegistry.this.notify(consumerUrl, listener, ZookeeperRegistry.this.toUrlsWithEmpty(consumerUrl, path, (List<String>) rawAddresses));

}

};

}

可以看到，如果某个 Consumer 的 providers 多次变更时，第一次变更会阻塞后续变更。并让 Zookeeper 合并多次通知，便能做到减少中间不必要的通知导致的 URL 重复创建。（此处还有一个问题需要思考，Consumer Watcher 阻塞时，会将所有接口的 providers 变更通知全部阻塞，导致其他接口通知被延迟接收。）