面向微服务应用的弹性测试方法

摘　要：当前，微服务架构已被腾讯、Netflix、Amazon 等公司用于线上环境，支持快速的迭代与灵活的发布。众多大型微服务应用依赖复杂的容错逻辑保障可用性。然而，一方面，微服务应用中的复杂交互关系使容错逻辑容易失效，另一方面，单元测试与集成测试对容错逻辑的测试效果并不理想。综上所述有针对性地进行弹性测试是十分必要的。面向微服务的弹性测试主要面临两个挑战：（1）为了寻找应用无法处理的故障场景（有效故障场景），需要尝试在不同的位置注入不同故障。面对巨大的探索空间，寻求自动高效的测试方法成为亟需解决的问题。（2）由于业务场景的不同，故障处理逻辑的具体内容也各不相同。为了辅助理解失效的容错行为，归纳常见的故障处理模式并确定分析方法成为亟需解决的另一个问题。本文基于约束求解技术实现了一种自动化故障注入方法。经实验验证，该方法只需要大约 20 次注入即可覆盖应用 Trainticket（包含 65 个服务）的所有故障场景。“冗余路径”的数量和复杂度对该方法影响不大，其效果远远好于随机算法。除此之外，论文对文献与流行框架中的常见故障处理模式进行了总结，提出对超时模式（Timeout）、重试模式（Retry）、船舱模式（Bulkhead）以及熔断模式（CircuitBreaker）的建模和自动化探测方法。经实验验证，该方法针对 Bookinfo 共探测出 18 条异常记录，并无错报和漏报现象，具有有效性。

关键词：　微服务；弹性测试；故障注入；故障处理模式；

第一章：引言

大型微服务应用为了实现”总是可用(Always-on)”的用户体验，在设计、实现与维护的过程中存在众多的故障处理逻辑。大型微服务应用通过服务降级、故障转移、超时重试等一系列方式对局部服务失效进行管控，保证核心部件能够正常提供服务，极大地提高了系统的可用性。除此之外，金融与军事领域对于执行结果的要求格外严苛。故障处理的设计与实现使应用以“可靠的”方式提供服务，避免了大量的经济与人员损失。综上所述，故障处理逻辑是大型微服务应用必不可少的一部分，对应用的容错能力产生深刻的影响[7]。

故障处理逻辑的目的在于帮助微服务应用能够“正确地”应对故障，降低应用受到的影响。然而，故障处理逻辑本身很有可能存在问题，导致微服务应用无法正常工作。在某些情况下，不当的故障处理逻辑将导致比没有故障处理逻辑时更糟糕的结果。2018 年中，AWS、微软 Azure 和谷歌云平台都经历了重大的云服务宕机事件[6]。事件持续时间从 20 分钟到 2 个小时不等，造成了严重的影响。2014 年至 2017 年之间，也发生了众多故障处理逻辑失效的事件[1][2][3][4][5]。其中 Twilio 公司由于 redis-master 的恢复策略不当，导致用户付款后不到账和系统重复扣款等经济问题。导致这些现象发生的原因主要有两点。

首先，微服务架构对功能模块进行划分并提倡单独开发、运行与维护，主要目的在于增加应用管理的灵活性。然而，应用的复杂度主要由业务逻辑决定，并不会由于采用微服务架构而减小。在微服务架构下，每个划分得到的微服务为某个单一的目的而存在。所以应用中各个微服务之间的交互关系变得庞大且复杂。一方面，杂乱的交互关系容易造成配置错误的现象。另一方面，交互内容由具体的业务场景决定，具有多样的功能性需求和非功能性需求。相关人员常常由于某些原因，例如对业务场景不熟悉或者盲目自信[5]，错误实现或配置重要的故障恢复逻辑。综上所述，应用的复杂度使交互关系变得异常复杂，治理微服务应用变得格外困难，很大程度上导致无效或不恰当的故障恢复逻辑。

其次，　单元测试与集成测试主要的测试对象是正常功能逻辑。一方面，单元测试和集

成测试在实施过程中并不会有意地多触发故障。应用中的众多故障处理逻辑无法触发，更无法测试。这将导致对于故障处理逻辑的覆盖率非常低。另一方面，即使单元测试和集成测试过程中触发故障处理逻辑，也并不会有针对性地基于当前的故障场景对应用的故障处理逻辑进行分析。这导致对于故障处理逻辑中问题的定位需要基于测试数据进行更多的分析与思考。综上所述，单元测试与集成测试对微服务应用中的故障恢复逻辑的测试效果并不好。

因此，研究面向微服务应用的弹性测试方法，并设计实现一个通用且开源的微服务自动化弹性测试工具，对于实现发现无效的故障恢复逻辑与提高大型微服务应用可用性具有十分重要的现实意义。

本文设计实现一种面向微服务应用的自动化故障注入方法，高效地发现应用无法处理的错误场景，并且调研微服务架构下常见的故障处理模式，针对不同的处理模式建模并提出自动化分析方法，最后基于以上关键技术，设计实现一种面向微服务应用的弹性测试工具，实现错误场景探测、处理模式分析、故障管理、调用链跟踪、系统状态监控等功能。

最后，本文通过应用 Trainticket 以及 Bookinfo 分别对故障注入方法和模式分析方法进行验证。实验结果表明，本文提出的故障注入方法可以在几乎不需人工干预的情况下，快速触发 Trainticket 中的所有故障场景。“冗余路径”的数量和复杂度对该方法效率影响不大。实验结果还表明，本文提出的模式分析方法可以有效地捕捉 Bookinfo 中的异常记录，并无漏报和错报的现象。

第二章：弹性测试方法

当前，面向微服务的弹性测试主要使用的是故障注入技术，面临注入位置、故障类型以及注入时机的求解问题。三个维度的不同赋值形成数量庞大的组合，几乎不可能以穷举的方式一一测试不同情况。例如，腾讯微信后台系统[8]在 2000 多台机器上运行超过 3000 个服务[9]。如果不考虑注入内容和注入时机，只从注入位置的维度计算不同情况的数量。每个服务有两种选择——注入或不注入故障，则共有2^{3000}情况。即使使用拥有巨大算力的太湖之光，当前情况下也无法在有限的时间内完成穷举。为了应对如此庞大的探索空间，已有的测试方法主要分为两类：人工注入和随机注入。一方面，前者严重依赖人工经验，虽然对少量局部接口具有较强的针对性，然而受限于对系统的有限认知能力，人工注入方法总体上的效果有待提升。除此之外，用户常根据具体业务场景指定注入内容，可扩展性较低。另一方面，随机方法缺少引导信息，不仅对探索空间的效率较低而且测试结果的质量也得不到保障。类似“在大部分服务中注入故障会导致应用崩溃”的结果对于发现、定位与修复问题几乎不具有指导意义。然而该类结果大量出现在不受限的随机测试结果中。所以设计并实现高效自动化的弹性测试工具是十分必要的。

除此之外，包括 Gremlin、Chaos Monkey 在内的大多已有测试工具能够发现容错的失效现象，但并不对错误逻辑的行为进行分析。测试人员需要根据不同的应用场景分析调用链、响应时间等监控数据，从而达到初步理解系统行为的目的。相关调研[20]表明，即使利用具有可视化功能的监控工具，分析数据并理解应用行为仍然是修复 Bug 过程中最消耗时间的步骤。所以设计与实现容错行为分析模块，辅助测试人员快速理解失效的容错行为是十分必要的。

综上所述，在面向微服务的弹性测试工具中加入高效自动的故障注入模块和容错行为分析模块是十分必要的。为了实现具有自动化注入能力和容错行为分析能力的弹性测试工具，本文面临以下挑战：

（1）微服务应用具有强烈的异构特性，不仅源码的语言和技术框架存在较大差异，而且服务直接依赖的运行环境也各有不同，因此和源码或操作系统等信息紧密耦合的弹性测试方法并不适用。

（2）微服务应用具有庞大的探索空间，需要寻求一种高效地自动地故障注入方法，既能尽量减轻对人工经验的依赖程度，又能够快速地根据探索历史做出分析，引导测试过程高效进行。

（3）微服务应用的故障处理逻辑和具体业务场景紧密耦合，需要根据容错行为的共性，对常见的故障处理模式进行分析与建模，设计简单高效地模式分析算法，辅助用户快速地理解应用处理故障的行为。

为了应对以上挑战，本文提出了一种面向微服务应用的弹性测试算法，不仅可以自动高效地寻找容错失效场景，还能够针对失效的容错行为进行自动分析。

对于挑战（1），虽然微服务应用具有强烈的异构特点，但服务之间消息交互的实现机制是类似的，即不同服务基于 HTTP 等标准轻量级协议进行信息交换。用户请求进入应用后产生一系列服务之间的调用过程，根据调用关系形成调用链。数据在调用链中流动时，其内容逐渐向目标结果转变。所以，利用调用链可以有效地解释特定结果的生成过程，从而作为自动求解注入位置的基础。

对于挑战（2），一方面为了对外提供“总是在线（Always-On）”的用户体验，在某些服务不可用后，应用会动态调整处理请求的过程，使用“冗余服务”完成相关功能。“冗余服务”在正常情况下并不接受用户请求，只有发生故障时才会出现在请求处理的过程中，所以对于故障注入而言，单独在“冗余服务”注入故障是无效的，不需要探测。另一方面，某些不重要的服务不可用后，用户仍然能够得到正常的结果，所以可以进一步精简用户请求的处理过程，用于指导后续故障注入过程。本文采用基于约束求解的故障注入方法实现上述剪枝过程，大量减少了需要测试的情况，提升了测试效率。

对于挑战（3），本文通过对相关技术[10][11][12][13]以及相关文献[14][15][16]的调研，建立了超时模式、重试模式、熔断模式以及船舱模式的相关模型，并基于模型提出了对应的探测算法，以不同的角度辅助用户理解应用行为。

根据以上关键思想，图 2.1 是本文提出的面向微服务应用的弹性测试方法的概览，主要分为 4 个部分：产生并收集调用链过程、求解注入位置过程、执行测试过程、分析应用行为过程。

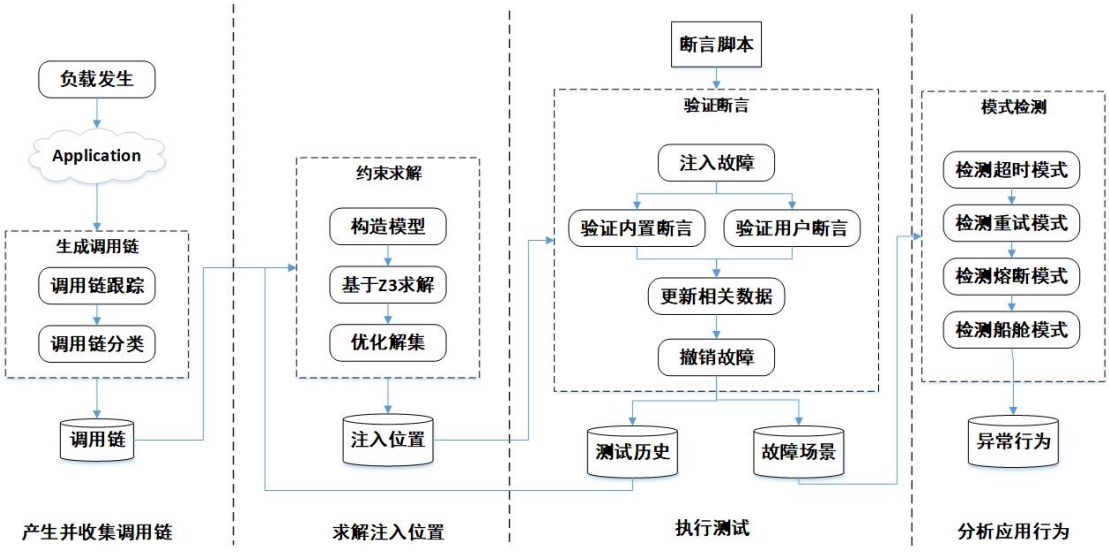

图 2.1 方法概览

本文提出的面向微服务的弹性测试算法如算法 2.1 所示，以待测应用作为输入，输出应用无法处理的故障场景信息以及对容错行为的分析结果。结合图 2.1 可以看到，本算法基本

分为四个关键的步骤：（1）产生并收集调用链的过程以待测应用为输入，跟踪不同用户请求并产生调用链信息，然后根据调用链的组成对请求分类，最后输出经过分类的调用链集合S_{trace}（2）求解注入位置过程的输入是调用链集合S_{trace}和测试的历史信息History，首先构造由约束表达式构成的约束模型，然后基于 Z3 约束求解器[17]计算解集，最后解码并根据历史信息剪枝，输出待测的故障注入位置IPS。（3）执行测试过程基于 IPS 注入故障，然后分别进行内置断言和用户断言的验证，记录无法处理的故障场景IPS_f并更新测试历史History，最后撤销故障。（4）根据IPS_f，从四个角度（超时模式、重试模式、熔断模式以及船舱模式）依次分析故障场景下应用的行为模式，输出不同故障场景下应用的异常行为。

测试过程的控制逻辑如算法 2.1 所示。步骤（1）完成后，算法得到调用链集合S_{trace}。针对每个调用链$trace_{req}$，步骤（2）（3）（4）以类似广度优先搜索的方式循环执行。首先针对当前调用链$trace_{req}$构造测试上下文context = $\langle S_{req}|History\rangle$，其中$S_{req}$表示请求 req 产生的所有调用链集合，History 表示针对请求 req 的历史测试结果。然后基于 context，步骤（2）生成待测注入位置集合 IPS。步骤（3）中对 IPS 中的每个故障注入位置逐个测试。未通过断言验证的故障信息将被记录并进入步骤（4）；通过验证的故障注入信息将用于更新 context。当 IPS 测试完毕后，算法将重新进入步骤（2）开始下一轮循环。当步骤（2）无法产生待测 IPS 位置时，针对当前请求 req 的测试过程结束，重新根据S_{trace}中其他的调用链构造测试上下文并重复上述测试过程。如果S_{trace}中的调用链全部经过测试，则算法结束。

算法 2.1. 自动故障注入算法

```
Function main(APP)
    Input: 待测微服务应用 APP
    Output: 无法处理的故障列表 f, 应用行为分析结果 r
    // 初始化测试结果集合
    f = r = Ø;
    // 步骤 1，产生并收集调用链，得到调用链集合 S_trace
    S_trace = produceAndCollectTrace(APP);
    // 步骤 2、3、4 循环执行
    For trace_req in S_trace
      context = < {trace_req} | Ø >;          // 构造测试上下文<S_req | History>
      // 步骤 2，得到故障注入位置集合 IPS
      IPS = SAT(trace_req, context);
      For IP in IPS
        // 步骤 3，逐个验证故障注入点
        If  verify(IP)
          Update(context);
        Else
          // 步骤 4，逐个验证故障注入点
          r = r ∪ Analysis(context);
          f = f ∪ IP;
    Return  f, r
```

如上文所述，本文算法由四个主要的部分。其中，第二部分、第三部分与第四部分是算法逻辑的主要部分，第一部分是算法输入数据的构造与收集部分。为使阐述过程更简洁，下

文将仅针对第二、三、四部分进一步阐述。

2.1 约束求解

大型微服务应用具有数以百计的微服务，注入故障的位置空间以幂函数的速度随着服务数量的增多而显著增大。逐个尝试所有注入位置是不可行的，有必要寻找一种高效自动地产生与优化算法。

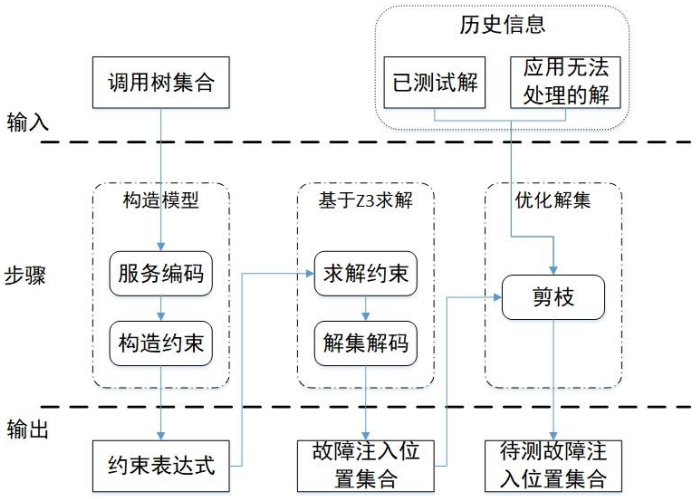

图 2.2 约束求解方法

本文基于调用链集合，通过约束求解的方法，实现了自动高效地求解注入位置的过程，如图 2.2 所示。自动化约束求解过程以调用链作为输入，具体包括三个部分：构造模型、基于 Z3 约束求解以及基于测试历史信息优化解集，最终得到待测故障注入位置的集合，作为注入与验证部分的输入。下面依次针对三个部分的实现细节进行阐述。为了更清晰地表达处理步骤，本文定义故障注入点概念如公式 2.1 所示。

$$IP_{APP} = \{service | service \in APP \cap var_{service} = True\} \quad \text{... (2.1)}$$

其中 APP 表示待测应用，$var_{service}$表示是否在服务 service 上注入故障。故障注入点(Inject Point,简称 IP)表示需要同时注入故障的服务集合，是约束求解的单个结果的形式化表示。约束求解结果的形式化表示为 IP 集合，即 IPS。

（1）构造模型

成熟的微服务应用能够为用户提供“一直在线”的体验，具有大量的冗余服务，例如 Netflix 公司的 AppBoot 应用[18]。冗余服务正常情况下并不存在于调用链中，只有在对应的目标服务发生故障后，调用链发生变化，冗余服务才会出现。所以在目标服务能够正常对外提供服务的情况下，只对冗余服务注入故障并不会影响请求的流量，即注入无效。需要同时覆盖冗余服务以及对应的目标服务。

基于以上思路，本文采用两个关键步骤构造约束条件，如算法 2.2 所示。首先保证每个调用树中至少包含一个故障，产生的约束条件可以表示为调用树中所有变量的析取形式。然后，同一类请求的每个调用树很有可能是彼此的冗余路径，应该同时注入故障，所以第二个步骤是将同一类请求的调用树所产生的析取表达式用合取符合连接，得到最终约束表达式。

算法 2.2. 约束表达式构造算法

Function *construct (Trees,vars)*

Input: 同一类请求的调用树集合 Trees, 服务变量集合 vars

Output: 约束表达式 constraint

```
constraint = Ø;
// 构造约束表达式
For t in Trees
   disjunction = Ø;
   For span in t
      disjunction = disjunction ∪ vars[span. tar]; // 收集单个调用树中的服务
   constratint = constraint ∩ join(disjunction); // 构造析取表达式和合取表达式
Return constraint;
```

（2） 基于 Z3 约束求解

Z3[17]是微软公司设计并实现的高效约束求解工具，其主要流程基于 DPLL 算法进行，运用冲突检测、布尔编码、搜索剪枝等多种技术，能够在短时间内高效地解决布尔可满足性问题。对于本文的应用场景，Z3 求解器可以快速判断约束表达式的可满足性并且给出一个满足约束的最简解，然而本文希望能够针对约束表达式求解所有的最简解，所以基于 Z3 的基本约束求解功能，本文提出一种增量式的 IPS 求解方法，其思想为通过已有解取反的形式逐步求出所有最简解。

给定约束表达式，本文利用 Z3 判断是否可满足。如果不可满足，则证明所有最简解都已得到；如果可满足，则对已有解编码，得到约束表达式，然后对表示取反并和原来的约束表达式用合取符号连接，表示目标解既需要满足原来的约束表达式，又不是已经得到的解。基于新的约束表达式重复进行上述步骤，所有解构成可以满足最初约束表达式的最简解集。

最后根据解集进行解码，每个解所包含的变量对应于单次注入需要同时注入故障的服务，即故障注入点（IP）。最终得到的结果为 IP 集合，简称 IPS。

（3）优化解集

虽然通过约束求解得到的解集数量相对探索空间已经非常小，但是仍然可以根据历史的测试结果对 IPS 进行剪枝。一方面已经测试过的 IP，无论能否发现逻辑错误，都不应该浪费资源重复测试；另一方面，当某个 IP 发现逻辑错误时，包含该 IP 的非最简解往往可以拆分或精简为多个最简解，为用户提供更精确地测试结果，所以没有必要测试包含有效 IP（能够发现逻辑错误的 IP，为了简洁，下文统一称该类 IP 为有效 IP）的 IP。例如，假设 $\{var_{API}\},\{var_{PLAYLIST}\},\{var_{RATINGS},var_{RATINGS_{CACHE}}\}$ 是最简解集合，对于 $IP=\{var_{API},var_{RATINGS_{CACHE}}\}$可以精简为$\{var_{API}\}$。本文将每个 IP 表示为一个字符串，其内容为服务名称按照字典序排序后的拼接结果。IP 之间的大小均为代表 IP 的字符串按照字典序比较的结果。下面本节分别介绍针对已测试 IP 和有效 IP 的剪枝方法。

针对已测试 IP 剪枝的过程假设存在两个 IP 集合，待测$IPS=\{str_{IP1},str_{IP2},\dots,str_{IPn}\}$和已测试$IPS'=\{str_{IP1},str_{IP2},\dots,str_{IPn}\}$，剪枝的过程即集合$IPS$与$IPS'$求差集的过程。本文借鉴数据库中表连接查询的实现，采用 HASH 匹配的算法，细节在此不再赘述。

针对有效 IP 的剪枝过程与针对已测试 IPS 的剪枝过程本质上相同，都是字符串列表的匹配过程，不同之处在于所有包含有效 IP 的待测 IP 都将被过滤,如公式 2.2 所示。

$$(\exists IP'\in IPS', str_{IP'}=str1)\wedge(\exists IP''\in IPS, str_{IP''}=str2\wedge str_{IP'}\in str_{IP''})$$
$$\rightarrow \nexists IP'', IP''\in R \qquad \dots (2.2)$$

子字符串匹配是一个传统问题，如果使用暴力搜索的算法，每对 IP 匹配过程的复杂度为$O(n^2)$，n 为 IP 字符串长度。本文采用 KMP 算法，以待测 IP 字符串str和有效 IP 字符串str'为输入，输出是否忽略该 IP 的测试。其核心思想在于计算str'的最长前缀，用于指导str和str'的匹配过程。该算法下，每对 IP 的匹配过程转换为$O(n)$，n 为字符串长度。算法细节

在此不再赘述。

2.2 验证断言

得到待测 IPS 后，算法对每个故障注入位置进行验证，目的在于判断当该位置发生故障后，应用是否仍然能够为用户返回正常响应。已有的验证方法大致可以分为两类：验证断言、推测用户行为。前者通过人工或内置的判断条件，验证容错执行结果，具有较高的准确性，但编写验证逻辑的过程比较耗时，可扩展性较差；后者适用于线上系统，可根据真实的用户行为推测应用状态，具有较高的真实性。由于缺少线上环境，本文采用第一类方法对故障注入效果进行判断，既提供通用的验证机制，又为测试人员提供自定义验证逻辑的能力。

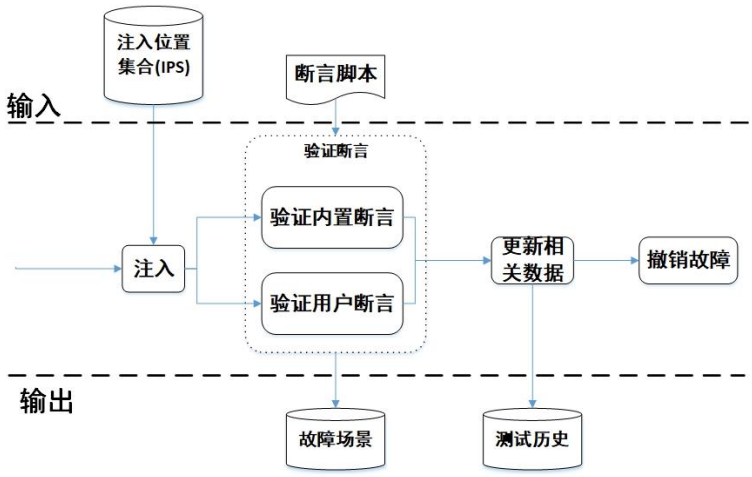

图 2.3 验证断言方法

验证的过程可以分为四个主要步骤，如图 2.3 所示。首先根据 IPS 注入故障，然后分别根据内置断言和用户断言验证响应是否正常。根据验证结果，本文将执行不同的数据更新操作。最后撤销故障，恢复应用正常行为。需要指出的是，为了使描述更简洁，图 2.3 中的处理流程是单个 IP 的验证过程。IPS 中的每个 IP 都将依次进行该过程。以下针对注入、验证断言、更新相关数据以及撤销故障等四个步骤进行阐述。

(1) 注入

一方面，尽管微服务具有强烈的异构特性，然而其交互方式常基于标准的轻量级协议实现，诸如：HTTP 协议、gRPC 协议等等。微服务之间的交互消息并不具有明显的异构特性，所以拦截与支配消息的过程不与服务逻辑的实现方式紧密耦合。另一方面，微服务应用以分布式系统的形式存在，所以故障服务行为的改变直接体现在对外交互信息的不同。本文通过控制服务间的交互方式达到故障注入的目的。该注入思路具有两点重要优势。一方面，可以有效地解决服务异构带来的挑战。另一方面，通过修改对消息的支配方式，可以快速地恢复故障服务。

本文对常见的故障场景进行了分析，发现异常的交互方式通常体现为两类负面效果的组合:返回错误信息(error)以及延迟过大(delay)，如表 2.1 所示。返回错误信息(error)指 HTTP 协议中包含相关错误描述信息，例如 500、503 等 HTTP 码[19]。本文将其抽象表示为 Abort(code),其中 code 表示返回的 HTTP 码，取值符合 RFC2616 标准。延迟过大(delay)指服务长时间无法完成请求过程，导致用户或上游服务长时间无法得到响应信息。本文将其抽象表示为Delay(duration)，其中 duration 表示响应的延长时间。基于 Abort 与 Delay 的组合，算法可以使目标服务的对外交互方式呈现异常状态，从而达到注入故障的目的。以 Overload 为例，使目标服务的 80%流量呈现 Delay(10s)，20%流量呈现 Abort(503)，则目标服务将呈

现过载状态。

表 2.1 常见故障场景

名称	效果
过载(Overload)	大量请求获得 503 响应，其余请求响应时间长
挂起(Hang)	请求长时间无响应（常以 10 分钟为单位）
断开连接(Disconnect)	请求得到 503、404 等非正常响应
崩溃(Abrupt Crash)	请求永久得不到 HTTP 响应
偶发故障 (Transient Crash)	少量请求得不到 HTTP 响应，大部分请求正常

本文构造了多种故障场景，如表 2.2 所示。在验证过程中，本文将依次尝试所有的故障场景，直至容错机制失效或容错机制可以处理所有故障场景。需要指出，根据相关实验与观察，本文构造一个自定义的故障场景：使目标服务的 50%流量具有 Abort(503)效果，其余 50%流量具有 Delay(20s)效果，表示故障服务无法处理半数请求，另一半请求的处理过程缓慢，需要 20s 时间。该故障场景在大部分情况下可以触发其他故障场景能够触发的容错失效现象，所以本文优先尝试该故障场景。

表 2.2 常见故障场景内容

名称	效果
自定义	Abort(500)-50% + Delay(20s)-50%
过载(Overload)	Abort(503)-80% + Delay(20s)-20%
挂起(Hang)	Delay(10min)-100%
断开连接(Disconnect)	Abort(404)-100%
崩溃(Abrupt Crash)	Abort(None)-100%
偶发故障 (Transient Crash)	Abort(500)-10%

（2） 断言验证

本文以用户或客户端收到的响应结果为分析对象，同时使用两种断言验证方式：内置断言验证与人工指定验证内容。前者提供了多种通用机制，对应用的响应结果进行验证；后者为判定标准提供了扩展能力。

内置的验证功能从多个角度结合不同的数据格式进行判断，具体如下：

首先依据响应时间验证断言。在实际情况中，响应时间对用户体验的影响存在 2-5-10 原则。等待时间超过 5 秒时，用户的体验非常差，继续等待请求完成的可能性将大大减少。本文利用得到的监控记录，以 5 秒为阈值，依据响应时间判定请求是否能被正常处理。如果大于 5 秒，则判定应用的故障处理逻辑存在问题。

然后根据响应码（HTTP CODE）验证断言。如果系统发生了严重的内部错误或者由于版本更新导致资源不存在，则用户往往得到包含 500 或者 404 等不正确 HTTP CODE 的响应。本文基于 HTTP CODE 对响应消息的语义进行判断，如果 HTTP CODE 以非 2XX 的形式存在，则判定响应内容不正确，即应用的故障处理逻辑存在问题。

最后针对 JSON 格式的数据分析语义。在很多情况下，应用并不会利用 HTTP CODE 表

达错误语义，而是将相关错误消息以 HTTP 有效载荷(load)的形式返回给客户端。该现象在以 JSON 格式交互的应用中格外普遍。本文通过相关调研发现，这类返回结果常依赖一个专门的字段描述响应是否成功。该字段名称具有高度的相似性，诸如：“success”、“suc”、“result”，并且字段通常出现在数据的顶层，数据类型为布尔型。基于以上观察结果，本文首先利用 HTTP 头“content-type”识别 JSON 数据格式，然后尝试基于字典抽取目标字段，最后通过该字段的值判断容错机制是否有效。用于抽取过程的字典如表 2.3 所示。

表 2.3 用于抽取响应结果的 Tags

Success, suc, result, Result, Res, res, Suc, successful, success

为了提供更灵活的验证方式，本文允许用户提供基于 python 语言开发的脚本。脚本以当前的注入内容为输入，对注入故障后的应用行为做出验证。执行结束后，脚本应返回测试结果，表明应用是否能够正常处理用户请求。

对于未能通过验证的故障场景，本文将进行记录并作为呈现给测试人员的主要结果之一。需要指出的是，如算法 2.1 所述，根据不同的验证结果，测试流程将进入不同的步骤。如果本次注入未导致应用异常，则基于当前的故障场景进行相关数据更新；如果本次注入导致应用异常，则基于当前的故障场景分析进行模式检测。

（3） 更新相关数据

如果应用在注入故障后仍然返回合理的响应结果，说明应用的故障处理逻辑有效。本小节针对正确容错的情况开展，通过调用链解释故障处理过程，为约束求解提供更多输入信息。

表 2.4 基于调用链的故障处理分类

编号	包含目标服务	包含备份服务
(1)	Y	Y
(2)	N	Y
(3)	N	N

根据调用链的组成，故障处理方式可以分为三种情况，如表 2.4 所示。（1）目标服务发生故障后，相关服务请求失败并更改调用目标，调用具有和目标服务类似功能的服务（称为备份服务或冗余服务）。该情况下，调用链既包含故障服务，又包含备份服务。（2）目标服务发生故障后，相关服务经历多次失败的尝试，不再尝试请求目标服务，直接请求备份服务。该情况下，调用链只包含备份服务，不包含目标服务（3）目标服务发生故障后，相关服务经历多次失败的尝试，不再请求包括目标服务和备份服务在内的任何服务。该情况下，调用链既不包含目标服务又不包含任何备份服务。

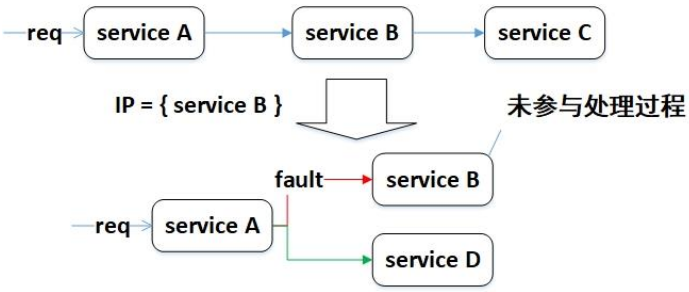

图 2.4 故障注入案例

对于情况（2）与情况（3），调用链中的每个服务都在请求的处理过程中起到了作用。

然而对于情况（1），由于已经在目标服务注入故障，目标服务无法正常提供相关功能，在请求的处理过程中并未起到作用，所以对于情况（1），调用链去掉目标服务后才能合理解释请求的处理过程。以图 2.4 为例，未注入前请求 req 由服务 service A、service B、service C 共同处理。在 service B 注入故障后，请求 req 实质上由 serviceA 与 service D 共同完成。虽然调用链中包含 serviceB，然而该服务对 req 的处理并未起到效果，应当在更新调用链时去除。

基于以上思想，本节利用 Jaeger 监控并收集注入故障后的调用链信息，然后去除调用链中的故障服务，最后更新调用链，为之后的约束求解提供更准确的输入信息。该过程对自动化注入算法至关重要。更新后的调用链是自动调节注入位置的重要依据。

（4） 撤销故障

算法通过管理交互信息实现故障注入，不需要修改应用源码和直接依赖的运行环境，所以撤销故障的过程不需要编译、重新发布或重启等相关步骤。恢复注入故障前的消息交互方式即可达到撤销故障的目的，具有代价小、恢复快速等优点，对基于不同技术实现的微服务具有通用性。

2.3 模式检测

相关调研[20]显示，对于微服务应用的 Debugging 过程通常包括七个步骤：（1）开发人员根据错误报告对应用的行为产生初步理解。（2）根据初步的理解搭建测试环境（3）在测试环境中复现 Bug，对故障的行为模式产生深入理解（4）确认故障的现象（5）确认故障服务（6）定位故障代码（7）修复 Bug。

在 Debgging 的过程中，步骤（1）是最消耗时间的过程之一。程序员需要分析大量的日志等监控信息，从而达到理解故障行为并初步判断故障原因的目的。诸如 Zipkin、Jaeger 等工具的可视化功能可以大幅提升该过程的效率，然而仍然严重消耗程序员的时间与精力。在该调研[20]中，利用可视化调用链技术，程序员平均花费 3 小时在步骤（1）中；利用可视化日志技术，程序员平均花费 7 小时在步骤（1）中；基于基础的日志分析，程序员平均花费 21 小时。有必要对于常见的故障现象进行自动化分析，辅助程序员对应用行为快速产生初步理解。本文的关注点是为何应用无法处理某些故障场景，所以基于无法处理的故障场景，对故障处理逻辑进行分析，加速程序员的 Debug 过程。

根据不同业务场景，故障处理的具体逻辑不同。然而这些处理逻辑符合某些模式。相关文献与技术表明，当前主要的行为模式包括超时模式、重试模式、熔断模式以及船舱模式。四个模式的关注点不同且并不具有明显的互斥性。故障处理的具体内容可以视为同时基于多个设计模式在业务场景下的设计与实现。对每个无法处理的故障场景，本文基于比对正常与异常情况下的相关数据，从四个模式的角度进行分析，辅助用户快速理解故障处理逻辑。下文分别对每个模式的分析方法进行阐述。

（1）超时模式分析

超时模式是最常见的故障处理模式，几乎出现在所有主流开发框架、服务管理平台与服务治理平台的实现中，为故障处理行为的分析提供了一个重要分析角度。超时模式的实现逻辑需要记录并检验请求时长。在规定的时间内，如果发出的请求未得到响应，则主动终止当前等待过程并判定响应失败。微服务应用通过超时模式的实现既保证了用户在有限时间内得到响应，又可以构建熔断模式、船舱模式等更复杂的恢复模式。

一方面，复杂的软件环境可能导致单个服务的真实超时阈值与预期不一致。另一方面，单个请求的处理涉及多个超时阈值不同的服务，导致请求的处理过程与预期的不同。超时阈值的不良设置不仅会导致请求处理的失败，还会导致微服务应用的状态不一致现象。为了说明该现象，本文设计实现了一个典型的案例并基于 Istio（1.0）环境部署。应用的服务以及服务间的交互如图 2.5 所示。用户请求首先到达处理服务。处理服务调用存储服务并阻塞，

等到存储服务的完成。然而由于处理服务的超时阈值过短，导致存储服务尚未处理完毕，处理服务就主动关闭了到存储服务的 HTTP 连接，告知用户存储失败。在 Istio 环境下，存储服务将继续完成存储工作，导致最终用户得到的结果与实际的执行结果不一致。综上所述，有必要从超时模式的角度辅助用户理解应用的故障处理行为。

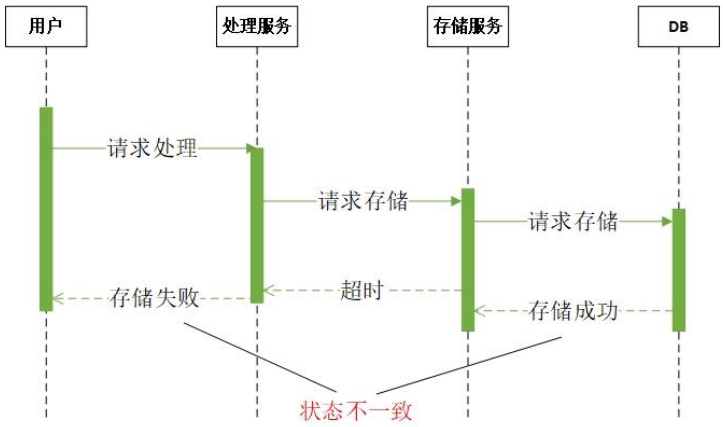

图 2.5 超时阈值不一致场景

通常情况下，程序员需要比较正常情况下服务的响应时间和异常情况下的响应时间，从而达到推测超时阈值的目的。对于复杂的调用链，该过程非常消耗时间和精力，所以本文自动化执行该过程，为程序员提供具有明显变化的响应时间数据，辅助程序员推测阈值冲突等问题。首先基于得到的有效注入位置，本文首先注入$\mathrm{Delay}(20s)$故障，然后根据调用链中的元数据统计相关服务的响应时间$t'_{src,tar}$，随后撤销故障并以同样的方式统计相关服务的响应时间$t_{src,tar}$。最后比对$t_{src,tar}$与$t'_{src,tar}$，输出异常的响应时间。

本文基于高斯分布判断$t'_{src,tar}$是否异常。如果$t'_{src,tar}$不在区间$[t_{src,tar}-\sigma, t_{src,tar}+\sigma]$，其中$\sigma$为标准差，本文认为源服务 src 到目标服务 tar 的请求并未受故障影响，否则认为请求的响应时间异常，很有可能和超时阈值的设置相关，所以输出相关信息。

（2）重试模式分析

重试模式是最常见的故障处理模式之一，在主流的开发框架、服务管理平台和服务治理平台的实现中都存在支持，为故障处理行为的分析提供了另一个重要分析角度。重试模式的实现逻辑为每隔一段时间（常为秒级别）对故障服务发送重复的请求。如果在某次重试后得到正常响应，则停止重试；如果在重试次数达到阈值后仍然无法获得正常响应，则不再重试，返回相关错误信息。微服务应用通过重试模式不仅可以有效应对临时故障（例如网络抖动造成的丢包），还可以构造熔断模式、船舱模式等更为复杂的故障恢复模式。

以重试的角度分析，真实行为与预期不符的原因也可以分为两方面。一方面，单个服务内复杂的软件层次可能导致过多的重试次数。例如，既在 Spring 层次配置重试次数为 3，又在业务代码中加入重试 2 次的逻辑，导致该服务最终将重试$3*2=6$次。另一方面，调用链中的不同服务也有可能产生类似的过多重试现象。综上所述，过多重试是不良重试模式导致的主要现象之一。

微服务应用在设计上具有幂等性原则，然而在实际的开发过程中，服务可能并没有幂等性的相关逻辑实现。重试模式将导致这类服务产生数据不一致的现象。 例如，当网络不佳时，请求支付的网络包需要较长时间到达支付服务。如果在等待的过程中，上游服务进行重试，则支付服务可能最终收到两个请求，导致多次支付。除此之外，非预期的过多重试也可能导致严重的性能问题。例如，在面对迅速增大的压力时，少量数据库访问失败并触发重试操作。此时，重试操作将造成额外的性能开销，导致数据库面临更大的访问压力，最终数据

库完全不可用。综上所述，有必要从重试模式的角度辅助用户理解应用的故障处理行为。

通常情况下，程序员需要比较正常情况下的调用链和异常情况下的调用链，从而达到了解重试次数的目的。对于复杂的调用链，该过程非常消耗时间和精力，所以本文自动化执行该过程，为程序员提供服务之间的重试次数，辅助程序员推测重试次数过多等问题。本文首先基于得到的有效注入位置，注入 Abort(500)故障，然后分别遍历正常情况下的调用链与异常情况下的调用链，计算服务间的调用次数，随后得到有明显变化的调用过程，即为重试过程，最后统计重试过程的次数并输出相关信息。

（3）熔断模式分析

熔断模式的核心思想在于两个方面。一方面，确认服务发生故障后，上游服务不再请求故障服务，而是进行降级处理，例如查询服务在数据库崩溃后，不再发送查询请求，直接返回本地缓存作为查询结果。另一方面，当服务从故障中恢复后，应该被上游服务重新使用。例如，网络故障恢复后，查询服务应当重新使用数据库中的数据作为响应。基于熔断模式的思想，微服务应用可以很好地对故障进行隔离，阻止级联故障的发生。

以熔断模式的视角观察，应用将在三个状态下不断转换。正常状态（闭合状态）下，服务对外提供服务；异常状态（开状态）下，上游服务执行降级逻辑，不请求故障服务；重试状态（半闭合状态）下，上游服务不断对目标服务发出尝试请求，试图发现目标服务恢复可用。为了确定熔断位置以及熔断触发的条件，程序员需要分析涉及不同状态的大量调用链。本文通过调用链比对的算法，自动化得出熔断位置并记录熔断触发前后的吞吐率，辅助开发者结合具体业务场景分析熔断行为。

算法 2.3. 确定熔断位置算法

Function *locate_circuit_breaker(t,t')*

Input: 正常情况下调用链 t，注入故障后调用链 t'

Output: 熔断位置 locations

```
// 初始化
locations = Ø;
// 获取根节点
root = getRoot(t);
root' = getRoot(t');
while root != null and root' != null
   bothChild, bothChild' = root.child ∩ root'.child;   // 待遍历子树
   part_locations = root.child – root'.child;   // 消失的子树为熔断位置
locations = locations ∪ part_locations;
locate_circuit_breaker(bothChild, bothChild'); // 递归求解
return locations;
```

基于有效的故障注入位置，本文首先获取正常情况下的调用链 t，然后注入 Abort(500)故障，获取相同查询在异常情况下的调用链t'，最后基于树型比对的算法计算熔断位置。如算法 2.3 所示，确定熔断位置的过程以正常情况下的调用链和注入故障后的调用链为输入，递归遍历两个调用链中都存在的服务调用，遍历过程中收集正常情况下存在但注入故障后不存在的子调用。消失的调用过程代表被熔断后消失的请求，其对应的上下游服务将以熔断位置的形式输出给开发者。除此之外，为了辅助开发者分析熔断的触发条件，工具还将简单对比熔断触发前与触发后的吞吐量变化。

（4）船舱模式分析

船舱模式又被称为舱壁模式，其核心思想在于隔离同一服务内不同接口的资源，从而避免共享资源被少量异常接口耗尽的情况发生。和前边的三个模式不同，船舱模式根据隔离资源的不同，实现方式完全不同。在流行框架中，使用船舱模式最多的资源是连接池。应用按照需求将单个或多个接口的连接池进行隔离，例如 Hystrix 提供的基于信号量和基于线程池的隔离策略。

程序员对于共享资源的分析通常需要观察一段时间内的请求速率变化，然后通过推测不同变化趋势之间的关系来确定接口对于共享资源的影响。本文对变化趋势进行简单的推测，辅助开发者了解共享行为。本文首先确定服务间的依赖关系，然后在下游服务注入 Delay(20s)故障。Delay(20s)使目标服务的响应时间变长，所以在一定时间内，上游服务到目标服务之间的连接数将增多，而吞吐率下降。监控上游服务到其他下游服务之间的吞吐率 throughput。监控的过程中保持稳定的并发用户请求。如果 thoughput 存在大幅降低的情况，则报告给开发者相关服务的接口之间彼此之间相互影响，可能存在共享资源的情况。

第三章：实验及结果分析

3.1 实验环境和实验对象

为了验证方法的有效性，本文分别针对约束求解方法以及模式检测方法进行实验。该实验的环境配置如表 3.1 所示。

表 3.1 实验环境配置

CPU	内存	操作系统	Java 版本
Intel-Xeon-E5-4607(2.6GHz)	128GB	Ubuntu18.04	1.8

实验对象分别选择 Trainticket 以及 Bookinfo。

（1）Trainticket

Trainticket[21]由复旦大学开发，是一个由 40+个服务组成的中型微服务应用，也是目前已知最大的开源微服务应用。其业务逻辑围绕订票、查票、检票、出票和后台管理等核心模块展开，业务场景比较全。除此之外，该应用通过 Java、NodeJS、Python 以及 Go 等多种语言实现，具有明显的异构特征。综上所述，Trainticket 具有一定的代表性，所以本文选择 Trainticket 作为主要的实验对象之一。

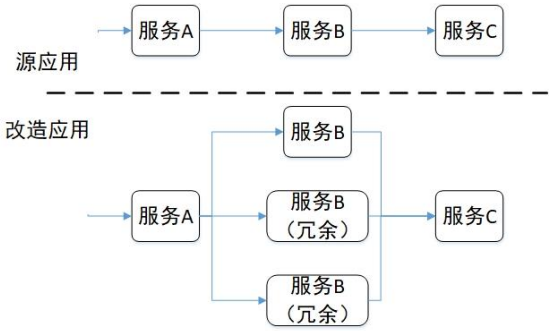

图 3.1 冗余构造方法

需要指出的是，Trainticket 应用几乎不具备故障处理逻辑，所以本文根据实验需要对 Trainticket 进行改造。改造的方式以图 3.1 中例子说明。对于图中请求，原始调用链依次包括服务 A、服务 B 以及服务 C。本文为服务 B 添加两个“冗余服务”，这两个冗余服务与服务 B 的业务逻辑完全相同。更改服务 A 的调用逻辑，在服务 B 不可用时调用冗余服务，冗

余服务不可用时调用冗余服务的冗余服务，以此类推。该方法使 Trainticket 具备一定故障处理能力，同时不会过多改变应用的原始逻辑。

（2）Bookinfo

Bookinfo[22]由 Google 公司开发，是一个由六个服务构成的小型微服务应用，被用于 Istio 框架的案例展示。其业务以展示为主，由 Python、Java、Ruby、NodeJS 等语言实现，如图 3.11 所示，具有典型的异构特点。除此之外，该服务将不同的功能独立地展示在界面的不同区域，便于故障处理逻辑的添加与调试。综上所述，本文选择 Bookinfo 作为主要的实验对象之一。

Booinfo 本身只有超时模式以及重试模式的实现。为了使 Bookinfo 更具有代表性，本文对 Bookinfo 的相关阈值进行调整，以硬编码方式添加熔断处理逻辑。

3.2 实验问题和评价方法

本节通过实验来验证面向微服务的自动化探测算法的有效性，实验主要回答以下问题：

（1）约束求解算法是否有效？

该问题中包含子问题：a，注入过程的效率如何；b，冗余服务的数量对效率有什么影响；c，冗余机制的复杂度对效率有什么影响。

为了回答该问题，本章实验使用 Trainticket 项目作为实验对象，分析了探索空间的规模，使用触发所有故障场景所需的最小注入次数衡量注入效率。另外，本部分实验以随机算法作为基准线，分别讨论了冗余服务的数量和冗余机制的复杂度对注入效率的影响。

（2）模式检测方法是否有效？

为了回答该问题，本部分实验使用 Bookinfo 项目作为实验对象，使用结果的正确比例衡量方法的有效性。正确性的判断基于第三方监控工具 Jaeger 和应用的真实行为方式进行。对于类似响应时间等具体数据，如果结果处于 Jaeger 监控数据的波动范围内，则认为正确；对于熔断位置、无法处理的故障场景等结果以应用的实际行为方式为标准。

3.3 约束求解方法实验

本文使用 Trainticket 项目截至到 2019 年 1 月 2 日的 master 版本源码。该应用共包含 65 个服务，其中 23 个是运行 MYSQL、MONGODB、REDIS 的数据库服务，42 个是运行业务逻辑的服务，为用户提供 10+个左右功能，涉及 50+个接口。本文挑选每个功能的重要接口作为判断对象，共计 27 个。

算法每次注入后，验证用户响应数据是否正常。经过多次注入后，算法如果已经使所有被选接口产生过异常，则认为已经覆盖所有故障场景，记录本次测试过程中的注入次数t_i,其中 i 为实验标号，从 1 开始。重复进行 N 次实验，取均值作为实验结果$\bar{t} = \sum_{i=1}^{N} t_i /N$。由于本文提出的测试方法是确定的过程，所以本文中取 N 为 10。最终 10 次实验的注入次数相同，均为 17 次。对于 Trainticket 而言，65 个服务将产生规模为2^{65}的探索空间。本文方法用 17 次注入即可覆盖该探索空间，测试效率较高。

大型微服务应用拥有数量众多的冗余服务。在某些服务发生故障后，相关服务将通过调用冗余服务的方式继续提供完整的业务逻辑。本实验分别讨论了冗余服务的数量和冗余机制的复杂程度对算法注入效率造成的影响。

本文首先为 Trainticket 添加了不同数量的冗余服务。然后以随机算法作为基准，分别对存在不同数量冗余服务的情况进行实验。触发异常需要同时在目标服务与冗余服务注入故障，所以随机算法每次选择两个服务作为注入位置，对一半流量注入 Abort(500)故障，对另一半流量注入 Delay(20s)故障，最后记录触发所有故障场景的最少注入次数。实验重复本算

法 10 次，重复随机算法 10000 次，取平均值作为最终结果。

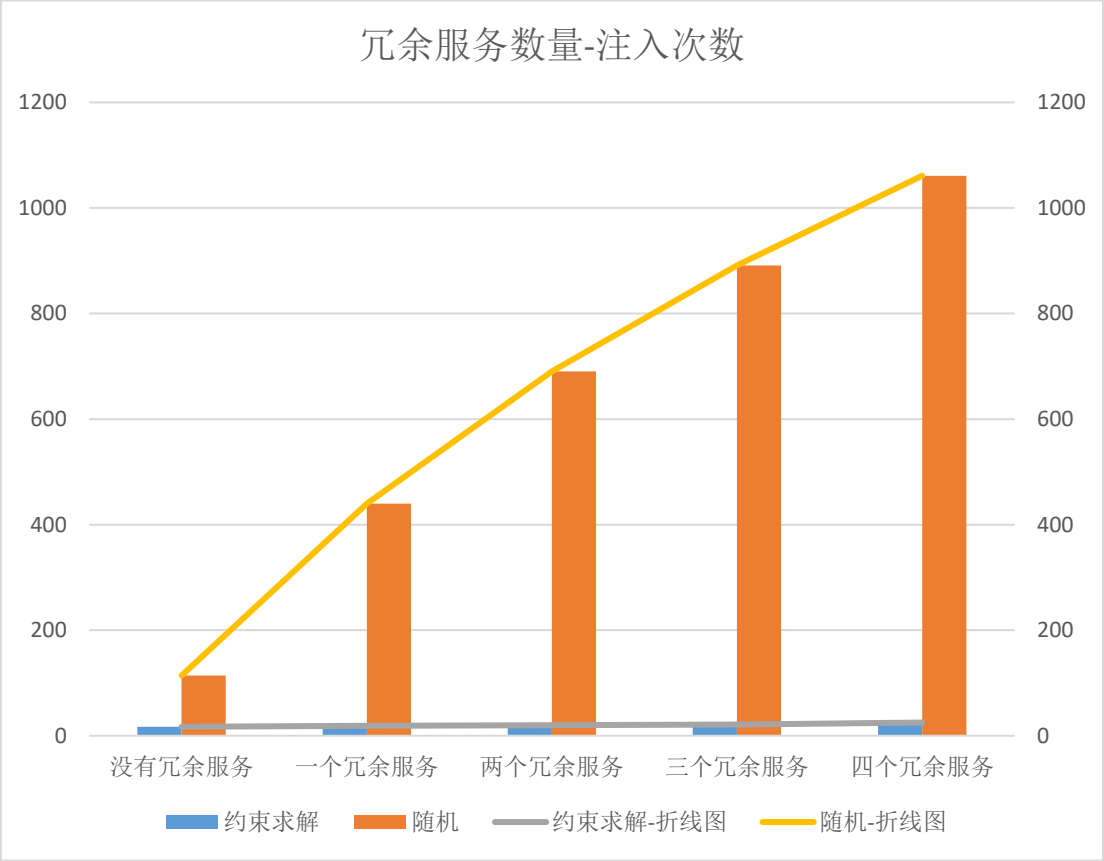

图 3.2 冗余数量对实验结果的影响

图 3.2 显示，当不存在冗余服务时，随机方法的需要进行 114 次注入才能够触发所有故障场景。约束求解算法只需要执行 17 次注入，相比之下测试效率高得多。随着冗余服务由 0 个增加逐渐增加到 4 个，随机算法的注入次数依次为 114，440，690，891，1061 次，呈现线性增长的方式，增幅在 200 到 300 之间。约束求解算法的注入次数依次为 17，19，20，21，25，相比之下几乎没有增长。充分说明了基于约束求解的注入算法的注入次数并不会随着冗余服务数量的增长而快速增长。

然后本文讨论了冗余机制的复杂度对算法效率的影响。为了更简洁的说明过程与结果，定义冗余机制的复杂度为触发故障场景所需的最小单次注入位置数量t_{min}。以图 3.1 为例，服务 B 为目标服务，$t_{min} = 3$。本文选择验证码服务(verification-code-service)作为目标服务，先后为其添加 0 个、1 个、2 个和 3 个冗余服务。在四种情况下，t_{min}分别为 1，2，3，4。随机算法根据t_{min}设置每次注入的位置数量。实验过程与上文相同。

图 3.3 显示，随机注入算法最初需要 114 次注入可触发所有故障场景。随着故障场景复杂度的增加，注入次数迅速增加，且增加速率也在不断增加。在故障场景需要同时注入四个位置时，随机算法的注入次数达到了 42154 次，探测效率已经无法接受。

图 3.4 显示，约束求解算法最初需要 17 次注入可触发所有故障场景。随着冗余机制复杂度的增加，注入次数以接近线性的趋势缓慢增加。最终在故障场景需要同时注入五个位置时，约束求解算法只需要 27 次注入即可触发所有故障场景。充分说明了基于约束求解的注入算法的注入次数并不会随着冗余机制复杂度的增长而快速增长。

出现以上实验结果的原因在于两个方面。一方面，对于随机算法而言，不同故障场景的触发在概率上是独立事件，所以故障场景的增多使随机算法的注入次数呈现线性趋势增加。

但当故障场景的触发条件变得复杂后，触发故障场景的概率将以幂函数的趋势降低，所以随机算法得注入次数会迅速增加而且增加速率也快速增加。另一方面，对于约束求解算法，求解过程基于调用链计算最简解，可以对探索空间进行有效地剪枝。无论冗余服务的数量还是冗余机制复杂度的增加都不会使得到的解集迅速变大，所以约束求解算法的注入次数并未产生明显变化。

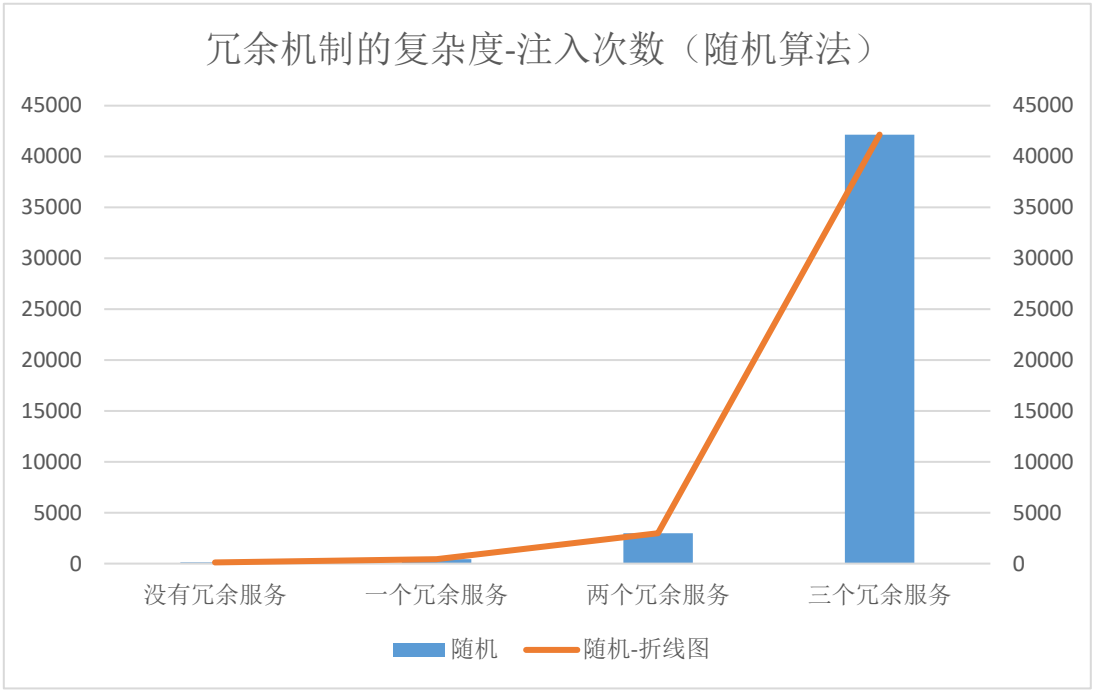

图 3.3 冗余复杂度对随机算法的影响

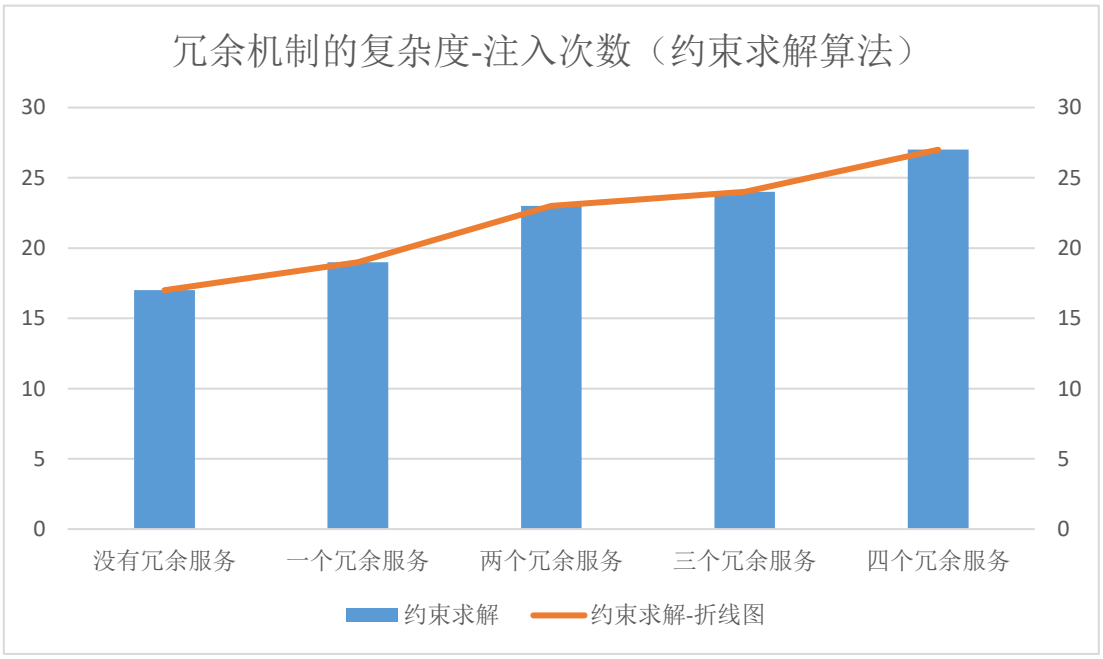

图 3.4 冗余复杂度对 SAT 算法的影响

综上所述，基于约束求解的注入算法可以高效地探测故障空间，发现应用无法处理的故障场景。当冗余服务的数量增多或冗余机制变得复杂时，算法仍然可以保持稳定。

3.3 模式探测方法实验

基于 Bookinfo 项目截至到 2019 年 1 月 2 日的 master 版本源码，本文进行的改造工作如图 3.5 所示。应用由 productpage、reviews1、reviews2、reviews3、reviews4 和 ratings 服务组成，具有四个典型的故障处理逻辑：（1）productpage 对 reviews3 的调用超时阈值为 5s，然而 review3 到 ratings 的调用超时阈值为 8s。这是一种典型的超时阈值设置冲突的现象，有可能导致应用状态不一致。（2）productpage 对 reviews2 调用存在最多 3 次重试（3）如果对 reviews3 调用失败后，productpage 将尝试调用 reviews2。如果仍然无法成功，则调用冗余服务 reviews1。（4）productpage 对 reviews4 的调用具有熔断实现。

图 0.5 Bookinfo 结构（修改）

本实验首先基于约束求解算法得到所有应用无法处理的故障场景作为输入，然后基于注入位置进行模式分析。输入信息如表 3.2 所示，共包含三个故障场景。以下根据不同模式分析实验结果的正确性。

表 0.2 有效故障场景

ID	接口	注入位置	注入内容	异常结果
1	/productpage	reviews{1~3}	Abort(500)	无评分信息
2	/productpage1	reviews4	&&	无评论信息
3	/productpage	ratings	Delay(20s)	无评论信息

（1）超时模式

超时模式分析的目的是将异常的响应时间呈现给用户。本实验以第三方工具 Jaeger 的监控结果作为标准，分析本方法得到结果的正确性。如表 3.3 所示，所有的响应数据均落在标准结果允许的误差范围内，说明本工具的超时模式分析结果比较准确。除此之外，结合拓扑结构，程序员可以很容易的通过第五条与第八条数据推测出阈值冲突问题，一定程度上说明了分析结果具有实用价值。

表 0.3 超时模式探测结果

场景 ID	源服务	目标服务	响应	标准结果

1	productpage	reviews3	5.002s	5.000s(±0.010)
1	productpage	reviews2	10.007s	10.000s(±0.010)
1	productpage	reviews1	10.000s	10.000s(±0.010)
2	productpage	reviews4	10.001s	10.000s(±0.010)
3	productpage	reviews3	5.000s	5.000s(±0.010)
3	productpage	reviews2	10.001s	10.000s(±0.010)
3	reviews2	ratings	8.010s	10.000s(±0.010)
3	reviews3	ratings	8.003s	10.000s(±0.010)

（2）重试模式

工具仅探测到一个重试现象，即 productpage 到 reviews2 的三次重试现象，符合应用的真实情况。在实验中并无错报与漏报现象，说明重试模式的分析结果比较准确。

（3）熔断模式

表 0.4 熔断模式探测结果

场景 ID	源服务	目标服务	结果
1	productpage	reviews3	缺失熔断机制
1	productpage	reviews2	缺失熔断机制
1	productpage	reviews1	缺失熔断机制
2	productpage	reviews4	于 reviews4 处熔断(0.5QPS)
3	productpage	reviews3	缺失熔断机制
3	productpage	reviews2	缺失熔断机制
3	reviews2	ratings	缺失熔断机制
3	reviews3	ratings	缺失熔断机制

熔断模式探测结果如表 3.4 所示，共在三个故障场景下检测到 7 次熔断缺失。算法于 reviews4 出检测到熔断机制，并且记录下探测流量大约为 0.5QPS。八次探测结果全部正确，并无漏报和错报现象。

（4）船舱模式

方法仅探测到一个资源共享现象：reviews{1~4}共享 productpage 的连接池资源。该结果符合实际情况。工具在实验中并无错报与漏报现象，说明船舱模式的分析结果比较准确。

综上所述，针对 bookinfo 应用中的三个故障场景，模式检测方法分析得到 18 条结果，其中包括 8 条超时模式分析结果、1 条重试模式分析结果、8 条熔断模式分析结果以及 1 条船舱模式分析结果。以 Jaeger 和应用的实际行为作为评价标准，测试结果全部正确且不存在漏报现象，一定程度上说明模式检测方法的有效性。

第四章：相关工作

本文主要针对微服务应用的故障处理逻辑进行测试，相关测研究工作以弹性测试工具为主。当前比较著名的自动化弹性测试工具有 Gremlin、ChaosMonkey。

Gremlin 由控制层与数据层组成。微服务应用与对应的代理共同构成数据层，以分布式的方式运行于底层的支撑平台上，例如 kubernetes、docker swarm。随着正常的请求流量与测试流量进入数据层，代理将拦截转发对应的流量并向控制层上报统计信息。控制层由脚本解释器(Recipe Translator)、故障下发组件(Failure Orchestrator)以及断言验证组件(Assertion Checker)组成。用户以 python 脚本的方式指定故障注入位置、内容以及对执行结果的预期结果。这些信息在数据层的脚本解释器中转为换 Gremlin 中的两种内部表示：注入信息和验证信息。故障注入组件将前者下发至数据层的代理中，用于使故障信息生效；断言验证组件利用代理上报的统计信息对执行结果进行验证，为用户提供验证结果。

注入内容方面，Gremlin 将复杂的应用场景归结为三种故障的组合：丢包故障(Abort)、延迟故障(Delay)以及失真故障(Modify)。开发者通过指定不同的故障参数、发生概率、发生位置，可以注入过载(Overload)、挂起(Hang)、崩溃(Crash)等等常见的故障场景。Gremlin 使开发者能够增量式地灵活地快速验证原型系统的弹性与开发者的猜测。

Chaosmonkey 使目标服务的部分或全部节点崩溃，选择方式完全随机。出于运行在线上环境的目的，对于注入范围的控制是 Chaosmonkey 的重要特性之一。以 kube-monkey 为例，提供了四种注入故障的模式：kill-all,fixed,random-max-percent 以及 fixed-percent。Kill-all 和 fixed-percent/fixed 相对应。前者将指定服务的所有实例从服务注册中心删除，模拟服务崩溃的情况；后者则对删除实例的数量进行限制。Fixed-percent 和 fixed 的区别在于前者按照百分比指定删除规模，后者按照绝对的数量指定删除规模。Random-max-percent 模式下，工具将随机删除一定数量实例，但是比例不会超过规定的上限。综上所述，利用 Chaosmonkey，测试人员可以快速针对线上系统进行弹性测试，注入内容为可控范围内节点失效。

Gremlin 需要用户指定断言脚本与注入内容。对于大规模的弹性测试而言，测试人员的工作量不可接受。除此之外，这种方式依赖于人工经验，对于复杂应用环境中的故障场景覆盖率并不高。ChaosMonkey 很难触发由多个服务故障时才能产生的故障场景，所以对故障处理逻辑的覆盖率不高。除此之外，两者都只能为测试人员提供容错失效场景，无法帮助测试人员理解无效的故障处理行为。本文提出的自动化故障注入方法可以快速定位应用无法处理的故障场景，并且针对常见的故障处理模式分析，辅助测试人员理解应用的行为。

第五章：结论及进一步工作

5.1 结论

本文设计实现了一种面向微服务应用的弹性测试工具，其中主要研究了自动化故障注入算法，和包括超时模式、重试模式、熔断模式和船舱模式在内的容错模式分析方法。论文主要贡献包括：

（1）本文提出一种面向微服务应用的自动化故障注入算法，实现了对故障处理逻辑的自动化测试，该方法不依赖人为指定注入位置和注入内容，且可同时根据调用链信息，自动更调整故障注入位置，大大减少了待测故障场景，有效提升了测试效率。除此之外，该方法采用基于 HTTP 拦截与支配的方式注入故障，不与待测服务的语言、框架、依赖的操作系统等信息耦合，适应微服务应用的异构特点。该方法基于约束求解技术生成覆盖调用链的最小注入点，并且能够根据测试结果动态构造约束，使注入点更精确。经实验验证，该方法在 Trainticket 应用中共注入 17 次故障即可覆盖2^{65}规模的搜索空间，而且该方法受冗余服务的

数量和冗余机制的复杂度影响较小，在存在大量冗余路径的场景下效果远远好于随机算法。该自动化故障注入算法组成了本文设计实现的面向微服务弹性测试工具的一部分。

（2）本文对常见的故障处理逻辑进行总结与分析，得到四种故障处理模式：超时模式、重试模式、熔断模式以及船舱模式。基于每个模式的核心思想，本文分别建立对应的模型，提出探测的方法，辅助测试人员了解应用的容错行为，提升 debug 效率，同时可以为相关研究工作或容错开发人员提供参考。

（3）基于以上研究成果，本文设计并实现了面向微服务应用的弹性测试工具，用户可以控制自动化测试过程，管理测试结果。除此之外，用户可以利用本工具手动管理故障并通过多种可视化技术对注入效果进行监控，利于快速灵活的掌握应用的容错能力。

5.2 研究展望

本文提出的面向微服务应用的弹性测试方法，实现了对故障注入位置以及注入内容的自动生成，可以在一定程度上帮助用户了解应用的弹性特征，定位无法处理的故障场景（有效故障场景）。其中在选择故障注入位置的时候主要考虑可以覆盖所有路径的最小注入点（最简解）。对于最简解的选择采用随机算法。根据观察，最简解的选择顺序一定程度上可以决定暴露有效故障场景的效率。所以在本文提出的基于约束求解的故障注入方法上，可以考虑基于拓扑结构等信息引入最简解优先级，从而达到快速暴露有效故障场景的目的。

另外本文选择的两个实验对象：Trainticket 以及 Bookinfo，自身并不具备复杂的故障处理行为。为了使实验的内容更加完整，本文采用了添加冗余服务的方式增加故障处理逻辑。该方法是真实情况的一种特例，无法代表全部故障处理逻辑的实现方式。根据相关观察，很多故障处理逻辑的实现通常伴随着业务逻辑的改变，例如，错误回调、响应缓存等。在保证尽量不修改应用自身业务逻辑的情况下，如何构造合理的有代表性的测试对象是个很有价值的研究问题。在自动化弹性测试的实验中集成类似研究成果，将使实验结果更具代表性。

参考文献

[1] “PARSE.LY. Kafkapocalypse: a postmortem on our service outage” [Online]. Available: http://blog.parsely.com/post/1738/kafkapocalypse/

[2] “CIRCLECI. DB performance issue” [Online]. Available: http://status.circleci.com/incidents/hr0mm9xmm3x6

[3] “BBC Online Outage on Saturday” [Online]. Available: http://www.bbc.co.uk/blogs/internet/entries/a37b0470-47d4-3991-82bb

[4] “SPOTIFY. Incident Management at Spotify” [Online]. Available: https://labs.spotify.com/2013/06/04/incident-management-at-spotify/

[5] “TWILIO. Billing Incident Post-Mortem: Breakdown, Analysis andRoot Cause” [Online]. Available: https://www.twilio.com/blog/2013/07/billing-incidentpost-mortem-breakdown-analysis-and-root-cause.html

[6] “2018 十大云主机宕机事件” [Online]. Available: https://cloud.tencent.com/developer/article/1380201

[7] Bogner J, Zimmermann A. Towards integrating microservices with adaptable enterprise architecture[C]//Enterprise Distributed Object Computing Workshop (EDOCW), 2016 IEEE 20th International. IEEE, 2016: 1-6.

[8] Wechat.Com, “Wechat,” 2018. [Online]. Available: https://www.wechat.com/

[9] H. Zhou, M. Chen, Q. Lin, Y.Wang, X. She, S. Liu, R. Gu, B. C. Ooi, and J. Yang, “Overload control for scaling wechat microservices,” in Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018, Carlsbad, CA, USA, October 11-13, 2018, 2018, pp. 149–161.

[10] “Sentinel: Sentinel of Your Application” [Online]. Available: https://github.com/alibaba/Sentinel

[11] “Fault tolerance library designed for functional programming” [Online]. Available: https://github.com/resilience4j/resilience4j

[12] Shneiderman, Ben. Response time and display rate in human performance with computers[J]. ACM Computing Surveys, 1984, 16(3):265-285.

[13] “Command pattern” [Online]. Available: https://en.wikipedia.org/wiki/Command_pattern

[14] 孙天竹，吴小兵. 采用故障注入技术提高系统可靠性[J]. 信息技术, 2004(6):85-86.

[15] Rajagopalan S, Sinha S. Prioritizing resiliency tests of microservices: U.S. Patent Application 15/229,958[P]. 2018-2-8.

[16] Montesi F , Weber J . Circuit Breakers, Discovery, and API Gateways in Microservices[J]. 2016.

[17] Moura L D , Nikolaj Bjørner. Z3: an efficient SMT solver[C]// International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2008.

[18] Alvaro, Peter, et al. "Automating failure testing research at internet scale." Proceedings of the Seventh ACM Symposium on Cloud Computing. ACM, 2016.

[19] Fielding R, Gettys J, Mogul J, et al. RFC 2616: Hypertext transfer protocol–HTTP/1.1[J]. 1999.

[20] Zhou X , Peng X , Xie T , et al. Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study[J]. IEEE Transactions on Software Engineering, PP(99):1-1

[21] “Trainticket github” [Online]. Available: https://github.com/FudanSELab/train-ticket

[22] “Bookinfo github” [Online]. Available: https://github.com/istio/istio/tree/master/samples/bookinfo