**Summary**

开源软件开发中依赖其他开源项目作为 submodule 非常普遍，理想情况下，这些我们只需要使用这些 submodule 即可。如果它们无法满足项目的需求时，我们也会尝试去贡献一些 feature/patch，这些特性由于项目迅速推进的需要，或者包含一些私有逻辑，往往会被积累而不能及时 merge 进 upstream。此时，如果这些 feature/patch 侵入性较大，且 upstream 更新较频繁，那么不可避免会产生大量的冲突。最近在 [**TiDB**](https://cn.pingcap.com/article/tags-3.html) 内核开发中，就遇到了这样的情况：为了能够及时 pick upstream 的 feature 或者 bugfix，我们需要频繁地进行 merge，但由此带来每次解决冲突和测试至少花费2人周。此外，也没有足够的测试来保证整个过程的安全性。显而易见，这些困难是4年前项目为了快速推进而积累下来的债务。在经历过多次痛苦后，我们认为已经到了必须做出改变的时候。我们花费了近半年的时间，对上下游的两个开源项目进行了重构，并极大程度地缓解了上述的问题。

通过本篇文章，你可以了解到：

如何在不对 upstream 注入私有逻辑的情况下，将下游 repo 的 delta 部分解耦为内聚性较高的独立模块。

解耦后，下游 repo 如何代理 upstream 的对外接口和读写配置逻辑。

如何渐进式地完成一个复杂软件工程的重构，如何设计测试以保证中间态的正确性。

关于维护一个频繁从 upstream merge 和 cherry-pick 的 git submodule 的一些方案。

**缘起**

TiDB 是 [**pingcap**](https://cn.pingcap.com/article/tags-1.html) 发起的开源分布式 [**HTAP**](https://cn.pingcap.com/article/tags-8.html) [**数据库**](https://cn.pingcap.com/article/tags-59.html)，TiDB 采用存储、计算分离的架构，由 TiDB Server、PD、TiKV、TiFlash 组成。其中 PD 是集群的元信息管理模块，处理诸如[**分布式事务**](https://cn.pingcap.com/article/tags-13.html) ID、Region 调度等全局事务；TiDB server 是一个无状态的计算引擎，用于分析 SQL 并生成查询计划；TiKV 是一个分布式的 KV 数据库，主要支撑 TP 负载；TiFlash 是基于自研列式存储的 MPP 引擎，用于给分析场景提供计算加速，适合 AP 负载。

TiDB 的存储节点 TiKV 和 TiFlash 使用 Raft 分布一致性同步协议在多个节点上同步数据。从 Raft 协议的视角，TiFlash 其实是一种特殊的 TiKV节点。为了避免重复实现 Raft，TiFlash 引入新组件 TiFlash Proxy 复用了 TiKV 这部分的代码，从而实现加入 TiKV Raft Group、向 TiFlash 同步数据等功能；并进行了大量修改以支持 TiFlash 的独有逻辑。本文要探讨的就是 TiFlash Proxy (后面简称为 Proxy)上对 TiKV 代码依赖的问题。在编译时，Proxy 产生一个 libtiflash\_proxy.so 动态库，在运行时被 TiFlash 装载，此后通过 Rust FFI 和 TiFlash 进行交互。具体的介绍可以参考文章《TiFlash Proxy 模块介绍 》。

如下图所示，长久以来，TiFlash Proxy 作为一个 TiKV fork 存在 https://github.com/pingcap/tidb-engine-ext，并被 TiFlash 以 git submodule 的方式利用。这样直接基于代码的利用方式，加上TiKV 本身的快速迭代，给 TiFlash 实现高效高质的开发升级带来较大压力。在下面的小节中，我们会详细论述这些问题。

**Merge TiKV 伴随巨大工作量和风险**

下面的链接展示了从 Proxy Merge TiKV 6.0 产生的变更 https://github.com/pingcap/tidb-engine-ext/pull/55/files，可以通过下面的命令复现

git fetch upstream 1e3f15fdb93a7ae41958d81c168d9e25ef3d4570 git fetch tikv release-6.0 git checkout -b test 1e3f15fdb93a7ae41958d81c168d9e25ef3d4570 git merge tikv/release-6.0 git merge --continue | wc -l

可以看到冲突文件达到 80 个，最终解冲突后的 pr 大小达到 700+ 个文件和 10k 行的 diff。这些文件中除去测试外，主要冲突达 193 处，位于 raftstore/tikv/server/test\_raftstore 等模块：

raftstore 是 TiKV 的核心组件，包含了基于 multi raft 的 KV store 的实现。为了实现写盘、行转列、pd worker 等机制，Proxy 需要修改 raftstore 的原本逻辑。所以尽管 Proxy 将自己的主要逻辑放在模块 engine\_store\_ffi 中，但该 mod 耦合在 raftstore 中，并且在很多类和函数中被以上下文的方式传递。

tikv 和 server 对 raftstore 进行了两次封装：在 TiKV 中被用于链接得到 tikv-server；而在 TiFlash 中被用于链接得到 libtiflash\_proxy.so。因此，Proxy 在编译目标，以及后续和 TiFlash 相互注册的额外逻辑上也进行了侵入式的修改。此外，由于 Proxy 在 TiKV 集群中的角色较为单一，所以在 server 中直接代码级地禁用了诸如 cdc、backup service、lock manager 等服务，这些也是冲突的来源。

test\_raftstore 为 raftstore 的测试框架，将在稍后探讨。

Proxy 对配置进行了增删改，只要 TiKV 也增加或修改了对应配置的默认值，则几乎必然导致冲突。配置项的冲突常常被忽视，但**一旦造成问题则很难排查**。特别地，代码一般都有确切的含义，解决冲突所以往往容易。而配置项的冲突除了冲突本身并不包含其他信息，因此通常往往不能从字面进行排查，只能查看提交历史或者文档来决定具体选择哪个版本。

升级过程还需要关注 Rust 和 Cargo 的一些问题。例如，每次升级后都会增加一些新的 feature，或者某些 crate 的 dependency 的 rev 会发生变动。这些变动可能不会以冲突的形式体现出来，但我们要对所有的 crate 适配这样的变动，否则至少会产生编译方面的问题。

目前 TiDB 内核发版间隔在 1-2 月左右，而一次升级可能就占用2人周，并带来潜在的风险。

**缺乏对 TiFlash 特性的独立和完整测试**

对 Proxy 的 PR 可能是 cherry-pick TiKV 的新特性，也可能是 Proxy 自己新增的特性。因此理想的测试应该分为两部分：第一部分用来测试改动是否会 break TiKV 的既有逻辑；第二部分用来测试改动是否符合 Proxy 部分的预期。

对于第一部分，由于 Proxy 直接修改 TiKV 代码，所以目前 code base 中已不含有纯 TiKV 的逻辑，很多 TiKV 的既有测试会被 Proxy 的改动 break 掉。我们只能手工适配若干重要的 test case 来保护 TiKV 的逻辑，这样的做法低效且局限。

对于第二部分，TiKV 使用 test\_raftstore 等组件作为测试和 mock 框架。由于 Proxy 直接修改 TiKV 代码，为了能正常启动运行，Proxy 需要对 test\_raftstore 进行改动，并新增 mock-engine-store 模拟 TiFlash 端的行为。这块的改动比较大，每次升级需要解决冲突。所以一些 Proxy 的独有特性，例如 TiFlash 的落盘模式、行转列、重启等部分，也缺少支持。

**Proxy 开发长期脱离 TiKV master**

正如前文所述，由于升级存在很大的开销和风险，Proxy 的不少版本都依赖很早期的 TiKV(如大约一年前的5.1)。首先这导致 Proxy 和 TiKV 彼此不 aware 对方的逻辑和特性。其次，随着后续 TiKV 对这些版本不再提供 bugfix，Proxy 各个版本的维护也会产生很大的困难。最后，为了整合发版测试中所有的 bugfix，我们通常要在 TiKV 发布版本 T 之后才能升级，所以 Proxy 理论上最好情况都会和 TiKV 相差至少一个版本。

**TiKV 不 aware Proxy 的逻辑**

由于 Proxy 的依赖 TiKV 核心模块，TiKV 的修改可能需要 Proxy 乃至 TiFlash 进行兼容。但目前的机制下无论从 TiKV 还是 Proxy 部分的单测都不能发现兼容性的问题，而会将问题推后到集成甚至回归测试中才能发现。理想情况下，对于 Proxy 的核心依赖应该在 TiKV 侧有单测保护；对于专有的逻辑，应该在 Proxy 侧的单测来保护，从而在 RFC 或者开发阶段解决 TiFlash(Proxy) 和 TiKV 的兼容性的问题。

例如，前段时间 TiKV 实现某特性，在集成测试时发现导致 TiFlash 无法正常运行，当时调试和修复花费了约2人日。

**对 Release/Hotfix 做 cherry-pick 有很大的难度和工作量**

由于 Proxy 基于较老的 TiKV 分支，这些分支可能已经不再维护，或者因为整体模块缺失/冲突而无法 cherry-pick。

除此之外，cp 操作会造成“PR”放大，即对于每个 release 需要至少两个 PR(Proxy 和 TiFlash)才能修复。对于 Hotfix 则可能需要4个 PR(Proxy 和 TiFlash 的 release，以及 Proxy 和 TiFlash 的 hotfix)。

我们也可以从中看到，对于一个快速迭代的项目来说，TiKV -fork-> Proxy -submodule-> TiFlash 这样的依赖链过于僵硬。在每次升级 Proxy 之后，我们都需要手动去更新 TiFlash 的 submodule。

**重构目标**

TiKV 是一个独立的项目。因此，尽管我们要从 Proxy 中剥离逻辑，但这些逻辑也不能被体现在 TiKV 的 code base 中。所以我们考虑的目标：

**目标1**

将 Proxy 的独有逻辑全部从 TiKV 中独立出来，这样 tidb-engine-ext 这个 repo 实际上包含了 TiKV master 的全套代码 + Proxy 的独有逻辑。每次 merge master 时，Proxy 的独有部分不会产生冲突，从而能提升效率。

**目标2**

基于第一个目标，我们还能更激进地彻底去掉 TiKV 代码乃至 tidb-engine-ext 这个 repo，Proxy 直接整合进去 TiFlash，通过 Cargo dependency 的方式依赖 TiKV，也就是通过指定 git repo 和 rev 的方式确定具体依赖的 TiKV。这样做的好处是不再需要维护 TiFlash 和 Proxy 的对应关系，减少 cherry-pick 和 hotfix 的工作量。但因为 code base 中不再有 TiKV 源码，不能通过 hack 源码方式来解决 TiKV 部分代码的 bug。当然，在紧急情况下，我们可以单独开辟一个 TiKV 的 Hotfix 来对应解决问题。

但无论如何，重构的主要工作量和难点都是在目标1上。所以目标1是主要考虑的对象。

**重构方案**

Tips：本文主要关注重构方案，关于 Proxy 和 TiKV 的相关机制，可以阅读《TiFlash Proxy 模块介绍》进行了解。

总体上，我们首先 Merge 最新的 TiKV release 到 Proxy，并以此为基础进行重构。在重构目标1完成后，再 Merge 最新的 TiKV master。

**新的 KvEngine**

TiKV 使用 KvEngine 来存储实际的 KV 数据和一些 Region 相关的 meta 信息，这在代码中被实现为一个 trait。所以 TiKV 在设计上允许实现对 KvEngine 有不同的实现。

TiKV 使用 RocksEngine (属于 TiKV 的 engine\_rocks 包)来实现 KvEngine，其底层基于 RocksDB。在 TiKV 中，写入的数据会暂存在 WriteBatch 中，并被批量写入 KvEngine 即 RocksEngine 中。但 TiFlash 底层是全新的列式引擎 DeltaTree，所以我们没有把真实数据写入 RocksEngine 的需要。对此，原先 Proxy 中处理方式删除所有对 RocksEngine 写入的代码，将它替换成写入 DeltaTree 中的 FFI 接口。

自然想到，可以引入一个 TiFlashEngine(属于新增的 engine\_tiflash 包)，和 RocksEngine 一样，它也 impl 了 KvEngine。但 TiFlashEngine 不会将普通的 KV 写入自持的 RocksEngine 中，而是直接通过 FFI 转发给 TiFlash 中的 DeltaTree。

此外，TiFlashEngine 还持有一个 RocksEngine。持有的 RocksEngine 一方面为了存储 meta 信息，例如一些和 Raft 状态机相关的字段在需要时就不需要去 TiFlash 侧去取了。此外，由于 TiKV 中对 KvEngine 的参数化上不完全，不少机制显式依赖 RocksEngine，于是持有的 RocksEngine 就可以派上用场。最后，在实现时，很多 KvEngine 部分的调用可以直接转发给持有的 RocksEngine 来做，所以我们可以避免自己实现不少逻辑。

但随后我们发现在 KvEngine 层转发会丢失 TiFlash 需要的 Raft 相关信息，例如某个 KV 它对应的 Raft Entry 的(index, term) 等，显然需要在 Raft 层进行这样的转发才行。因此目前 TiFlashEngine 主要用来屏蔽掉对 RocksEngine 写入真实数据。

一般来说，随着 TiKV 的升级迭代，KvEngine 的接口可能发生变动。虽然对于大多数较小的改动，可以直接转发来处理，但当较大的改动发生时，根据最新版本的 RocksEngine 修改得到 TiFlashEngine 往往更方便。

**Observer**

在 raftstore 中提供了 observer 机制以侦测和响应 raftstore 中的各种事件。在 raftstore 初始化时，各个 Observer 会向 raftstore 注册自己需要监听的事件。当这些事件被实际触发后，raftstore 会通知所有注册该事件的 Observer。容易想到，如果我们在 raftstore 处理 apply 数据的过程中注册恰当的 Observer，就可以通过它们将写入通过 FFI 传递给 TiFlash。

TiKV 中已经包含一些 Observer，但不足以适配 Proxy 的场景。例如某些 Observer 在 apply 过程之外，所以我们并不能从中获得足够的信息。此外，Proxy 也不能将必要的副作用传递回去从而影响 apply 过程。

为此，我们需要在 TiKV 端新支持一系列 Observer 函数，以 pre(post)\_apply\_snapshot 为例进行说明。原先的实现侵入性较大：在线程 1 收到文件后，需要将其提交到 raftstore 中的一个通用线程池中，在该线程池中会通过 FFI 继续转发给 TiFlash 进行处理；线程 2 中会定时尝试取出 raftstore 队头，并通知 TiFlash 处理对应的 SST 文件。

可以看到，原先的逻辑中，raftstore 和 Proxy 的逻辑之间存在较强的耦合。通过 Observer，我们可以将 Proxy 的逻辑(红框)彻底解耦。如下图所示，我们首先将预处理逻辑的线程池移动到 TiFlashObserver 中，这样就可以同时从 raftstore 中剥离相关的 FFI 调用。接着我们为线程 1 和线程 2 的逻辑分别添加一个 Observer 函数，把事件转发到 TiFlashObserver 中。

除了 pre(post)\_apply\_snapshot，我们还新增或适配了 pre(post)\_exec\_cmd、on\_region\_changed、on\_compute\_engine\_size、on\_empty\_cmd、pre\_persist 等 Observer，在此我们不再详细展开，有兴趣的同学可以查看 Proxy 最新版本的 pr 和源码。

Proxy 中将原先嵌在 raftstore 中的 engine\_store\_ffi 模块独立出来，在其中定义了 TiFlashObserver 去实现上述这些 Observer。通过 KvEngine 和 Observer， 我们最终将 TiKV 的 raftstore 模块中大部分逻辑解耦出来。

**组装模块**

除去 raftstore 模块，Proxy 还对其他模块进行了侵入性的定制。例如我们希望把 TiFlash 的一些状态推送到状态服务器 Status Server 中；或者在启动 server 时 Proxy 希望通过各种方式去禁用掉一些服务等。这些需求往往过于特定于 TiFlash，通过上面的方案“开洞”会破坏 TiKV 的逻辑。

此外，对 Proxy 自身引进的模块，我们也需要从 TiKV 中取出并封装，从而减少耦合。

对此，新 Proxy 的解决办法是，全部重新组装这些模块。现在的工程架构如下所示，其中：

gen-proxy-ffi 模块从 C++ 头文件生成对应的 Rust FFI 接口代码到 engine\_store\_ffi 模块中。在生成时，我们根据头文件内容求出 md5 值，从而在运行时校验版本。

raftstore-proxy 是一个 c wrapper，负责将 proxy\_server 封装成 libtiflash\_proxy.so 这个 C dynamic library。

proxy\_server 整合了 TiKV 的 server 模块：

初始化 TiFlashEngine(而不是 RocksEngine) 和 TiFlashObserver。

使用独立的 TiKVServer 和 StatusServer。这方便我们移除 cdc、backup\_stream、lock manager 等 Proxy 不需要的服务。

engine\_store\_ffi 从 raftstore 中彻底剥离，并由 mod 独立为 crate。在拥有最初的 FFI 模块以及从 raftstore 剥离的代码外，该模块还包含 TiFlashObserver 的具体实现。因此新 Proxy 具有很好的内聚性。

engine\_tiflash 即对 KvEngine 的具体实现。因为 engine\_store\_ffi 和 raftstore 都对 engine\_tiflash 有依赖关系，所以在实现时，该模块中会尽量避免涉及和 TiFlash 直接相关的逻辑，从而避免构成反向依赖。

**配置**

在上面拆分模块后，最自然的解决办法是将 Proxy 的配置独立成到单独的配置文件中，但实际我们不能这样做。原因首先是 Proxy 和 TiKV 的配置存在很多不兼容的情况，具体体现在：

Proxy 新增了一些自己独有的配置

例如行转列的线程池大小参数，它服务于 Proxy 自己独有的机制。

Proxy 需要 TiKV 的某些既有配置的默认值

例如用于 IngestSST 的线程池大小，它在 TiKV 和 Proxy 中都有作用，但因为两者场景不同，所以需要选取不同的默认值。

Proxy 需要 TiKV 配置项的最终值

例如 TiKV 会根据用户输入生成配置项中的 Engine Label 字段，但 Proxy 需要在最终的配置中强制设置 Engine Label 为 "tiflash"。

其次，旧版本的 Proxy 在实现时直接修改 TikvConfig 的定义，并也以此读取配置文件 tiflash-learner.toml。这导致 Proxy 和 TiKV 端的配置不兼容了。但这样在集群升级时会按照 TiKV 的逻辑去读取解析 Proxy 修改后的 TikvConfig，从而无法处理上述三种不兼容问题。

为了解决兼容性问题，保证升级后能够延续读取旧集群的配置数据，我们提出一种“配置双读”的方案：

对于 TikvConfig，我们平行建立一个 ProxyConfig，包含

Proxy 独有的配置。

Proxy 需要 override TiKV 默认值的配置(在对应 impl Default 中修改)。

在加载配置前，分别按照 TikvConfig 和 ProxyConfig 解析一遍配置文件

对于 TikvConfig 和 ProxyConfig 无法解析的配置项，报错或者忽略。

对于只有 TikvConfig/ProxyConfig 能解析的配置项，分别解析到 TikvConfig/ProxyConfig 中。

对于 TikvConfig/ProxyConfig 都能解析的配置项，视为潜在配置冲突，交由 address\_proxy\_config 处理：

如果两者解析结果不一致，说明是按照不同默认值解析出的缺省配置，ProxyConfig 会去覆盖 TikvConfig。

如果两者解析结果一致则不处理。

此外，对于最后一种不兼容情况，可以在这里直接修改 TikvConfig 作为最终配置。

通过上述方案，用户只需要在 ProxyConfig 中声明，或是在 address\_proxy\_config 中写上一行代码，即可解决 TiKV 和 Proxy 的配置冲突以及可能带来的升级风险。

**使 Proxy 直接依赖 TiKV**

在目标1完成之后，我们得到了一份和 TiKV 一样的代码，以及独立出来的 Proxy 的修改。我们目前正在借助于 Cargo 包管理机制，使得 Proxy 直接依赖 TiKV，从而在 code base 中去掉 TiKV 的代码，也就是之前计划的目标2。整个过程分为两步：

将 Proxy 独有模块中的依赖项从 path 形式改写成 git 形式，并指定对应的 rev 版本

  以 raftstore 的依赖为例，原先依赖于 ../raftstore 中的代码，但由于这段代码等同 b5329ee 这个 commit，所以我们可以将这个依赖转写为 git 形式。

raftstore = { path = "../raftstore", default-features = false } === raftstore = { git = "https://github.com/tikv/tikv.git", rev = "b5329ee00837b2b6e687129b2cdbfc77842c053b", default-features = false }

修改工程根目录的 Cargo.toml，将它变为一个 virtual workspace

一个 Rust Cargo 项目可以从上到下分为 workspace、crate、mod 等多个层级。TiKV 在根目录的 Cargo.toml 的 Workspace 下包含诸如 engine\_rocks、raftstore 等多个 crates。现在我们已经将这些 crates 的依赖改写为 git 形式了，但还有一个特殊的 crate 也要处理，那就是根目录的 Cargo.toml 中定义的一个 tikv 这个 crate，我们也应该以 git 的形式去依赖它。这样，根目录下的 Cargo.toml 中就只包含 workspace 的定义了，即 virtual workspace。

**实施**

**渐进式的修改**

重构的代码变动很大，整体替换的方式风险难以估量。所以我们采取渐进式的方案，也就是分为多个 PR 提交，每个 PR 只尝试新增一小部分接口。在 TiKV 同学 Approve 某个 PR 后，再 cherry-pick 回 Proxy，并替换掉和该接口相关的 hard code 代码。每个接口适配经历的流程可以分为：

在 Proxy 上实现一个 Prototype。

将 Prototype 对 TiKV 的修改(称为接口) Cherry-pick 到 TiKV master，并提交 PR。

在 TiKV merge 后，将 merge commit cherry pick 到 Proxy 分支。因为最终版本可能有较大改动，所以这里可能需要解冲突。

迁移原先的 hard code 到 Proxy 侧实现新增的接口。

恢复原先的 hard code 到对应的 TiKV 版本。

对于整个过程中测试的兼容在下面讨论。

**测试**

工欲善其事，必先利其器。从前 Proxy 的一个痛点是没有独立和完整的测试。为此，我们引入了 new-mock-engine-store，使得：新 Proxy 能够跑通来自 TiKV 的测试，从而验证 Proxy 的修改不会影响 TiKV 的逻辑；并且有一个 Proxy 特有逻辑的测试，来验证 Proxy 包括落盘、重启之内的完整链路。new-mock-engine-store 完善了 mock-engine-store 中对 TiFlash 的 mock 中的欠缺部分，并整合了从前对 test\_raftstore 的修改，从而使得整个测试与 test\_raftstore 解耦。

因为整个重构是渐进式替换的，所以我们的测试要匹配这个过程。为此我们提出了一个 mixed mode 测试的方案：

初始状态：code base 中全部为旧 Proxy 的代码逻辑，测试只包含从旧 Proxy 从 TiKV migrate 来的测试 tests/tikv。

引入 new-mock-engine-store 和 tests/proxy 作为新 Proxy 的测试框架和内容，给它们打上 compat\_new\_proxy 这个 feature。

在迁移的初期，我们对 code base 中的修改打上 compat\_new\_proxy。在测试环境中，这些逻辑只对 tests/proxy 生效，因而可以验证迁移后的逻辑，并不影响 tests/tikv。

随着迁移的进行，Proxy 特有逻辑部分的测试被慢慢转移到 tests/proxy 中；而由于我们从 TiKV 中解耦了 Proxy 的特有逻辑，也可以逐步去除 tests/tikv 中手动适配的代码，使得它们恢复在 TiKV 中原来的样子。

在迁移的后期，我们已经可以将不少模块 checkout 回 TiKV 的代码了，在这些代码中保留 compat\_new\_proxy 就没有必要。因此我们删除 compat\_new\_proxy，对于残余模块中需要保持旧 Proxy 代码逻辑的部分，打上 compat\_old\_proxy 这个 feature。这样带上 compat\_old\_proxy，我们依然可以运行 tests/tikv 测试。

迁移结束意味着 Proxy 特有逻辑被彻底被移出 TiKV。最终我们得到了 tests/proxy 用于测试 Proxy 特有逻辑，而 tests/tikv 用于测试 TiKV 原始逻辑。

Mixed mode 的测试机制保证了在整个迁移过程中的大部分新老状态代码可以实现测试的覆盖，帮我们在单测阶段阻拦了大部分 bug。在新老代码并存阶段，只漏了一个 bug 到回归测试中。

**有关 Git branch 和 PR 治理**

**升级策略**

Git merge 主要包含 squash merge、merge commit 和 rebase 三种方式，一般我们 Merge PR 会使用 squash merge 的方案，将自己分支上的修改 Squash 成一个 commit 提交到 upstream/master 上。在 Proxy 升级对齐 TiKV 方面，我们使用 merge commit 的方式进行升级，原因是这样的方式能够更好保留 TiKV master 上的提交历史，从而减少在后续 merge 时候的冲突。

**Cherry pick 策略**

在目标2实现之前，我们仍然保留 tidb-engine-ext 这个 repo 作为 TiFlash 的 submodule，如上所述，这给 cp 和 PR 带来麻烦。我们从中发现几点优化方向，并形成 https://github.com/CalvinNeo/ghcp 来辅助更新 Proxy 到 TiFlash 的工作。

首先，可以将对 Proxy 的更新大概分为 TiKV/Proxy 的 Bugfix/Feature 四类：

对于 TiKV 的两类，Proxy 是被动进行更新，所以对应的 issue 和 PR review 体现在 TiKV 即可。

对于 Proxy 的 Bugfix，一定会关联 TiFlash 的 Bug Issue，并在 Proxy 端进行 PR review。

对于 Proxy 的 Feature

如果涉及 TiFlash 的功能，则会在 TiFlash 创建 Issue。

如果局限于 Proxy 自身的优化，则使用一个公共的 Issue trace board。

在对以上来源进行归类后，可以发现在 Proxy 的 PR 被 Approve 和 Merge 后，除非涉及 TiFlash 本身的 Issue，需要同时 attach TiFlash 部分的代码，对于其他的情况我们都可以自动化地生成到 TiFlash 更新的 PR，并跳过 review 过程。因此在该 repo 中，我们提供了一个脚本来自动化这一过程。

往 Proxy 的历史版本中 Cherry-pick 一些 TiKV 的修复也是十分令人头痛的事情。在 Cherry-pick 到较低 Proxy 版本时，我们发现因为其依赖较老的 TiKV 缺乏其他一些机制，导致很难让代码正常运转。对此我们的方案是，如果 TiKV 已经事先将该修复从 master 上 cp 到 Proxy 依赖的 release 版本，那么就可以让每个版本的 Proxy 从对应版本上 cp，从而自动地获取 TiKV 的 Cherry-pick 方案。这一方案存在两个问题：

首先是在版本很多的情况下，存在多个 PR 和 commit，手动处理很容易出错，为此我们同样准备了一个脚本，给定一个 TiKV master PR 和 Proxy 的待修复版本，通过 Github GraphQL 可以获取所需的所有信息，并引导逐版本修复，维护者需要做的只是在确认解决冲突后提交即可。

其次是一些修复并没有 cp 到对应的 release 分支，这大大限制了上述方案的应用场景。对此，我们可能需要使用一些降级的方案，例如从上一个被修复的 Proxy 版本 cp，但关于这个目前还有待研究。

**重构效果**

在目标1重构完成，也就是彻底将 Proxy 和 TiKV 代码解耦并恢复 TiKV 原本代码后，我们 merge TiKV master。可以从 after-refactor-demo 分支尝试 merge TiKV 6.3 来复现这次操作。

这次 merge 和上次时隔4个月，2个版本，但冲突少了很多。本次 merge TiKV 代码只产生了有限几个冲突，并基本是因为 TiKV 对代码重新进行了 code format 和部分重构。但我们可以基于 theirs 策略来解决这些冲突，即对于所有的冲突，都选取 TiKV 侧的修改。我们可以直接通过 git checkout 来完成这样的工作，也可以手动进行修改，无论选择哪种方案，基本都能在两小时内解决问题，相比之前至少2人周有了很大的改善。

目前我们正在着手实现目标2，也就是将 TiKV 部分的代码彻底从 Proxy 中移除，变为 Cargo 依赖。在这一部分做完后，绝大多数情况下只需要更新 TiKV 的 rev 版本即可。