





「ではみなさん、そういうふうに川だと云われたり、乳の流れたあとだと云われたりしていたこのほんやりと白いものがほんとうは何かご承知ですか。」先生は、黒板に吊した大きな黒い星座の図の、上から下へ白くけぶった銀河帯のようなところを指しながら、みんなに問をかけました。

「ジョバンニさん。あなたはわかっているのでしょうか。」

先生はしばらく困ったようすでしたが、眼をカムバネルラの方へ向けて、

「このほんやりと白い銀河を大きないい望遠鏡で見ますと、もうたくさんの小さな星に見えるのです。ジョバンニさんそうでしょう。」

先生は意外なようにしばらくじっとカムバネルラを見ていましたが、急いで「では。よし。」と云いながら、自分で星図を指しました。

「大きな望遠鏡で銀河をよく調べると銀河は大体何でしょう。」

「このほんやりと白い銀河を大きないい望遠鏡で見ますと、もうたくさんの小さな星に見えるのです。ジョバンニさんそうでしょう。」

やっぱり星だとジョバンニは思い出ましたがこんどもすぐに答えることができませんでした。

「このほんやりと白い銀河を大きないい望遠鏡で見ますと、もうたくさんの小さな星に見えるのです。ジョバンニさんそうでしょう。」

やっぱり星だとジョバンニは思い出ましたがこんどもすぐに答えることができました。



パッチを書く前に	1
パッチの作成	5
パッチを書いたあと	11
Object Storage の概要	15
Swift の運用と管理	29







Chapter 1. パッチを書く前に



OpenStack の開発に参加するためには、最初に少し準備が必要になります。まずは OpenStack 開発の概要を述べていき ます。

ソースコードリポジトリ

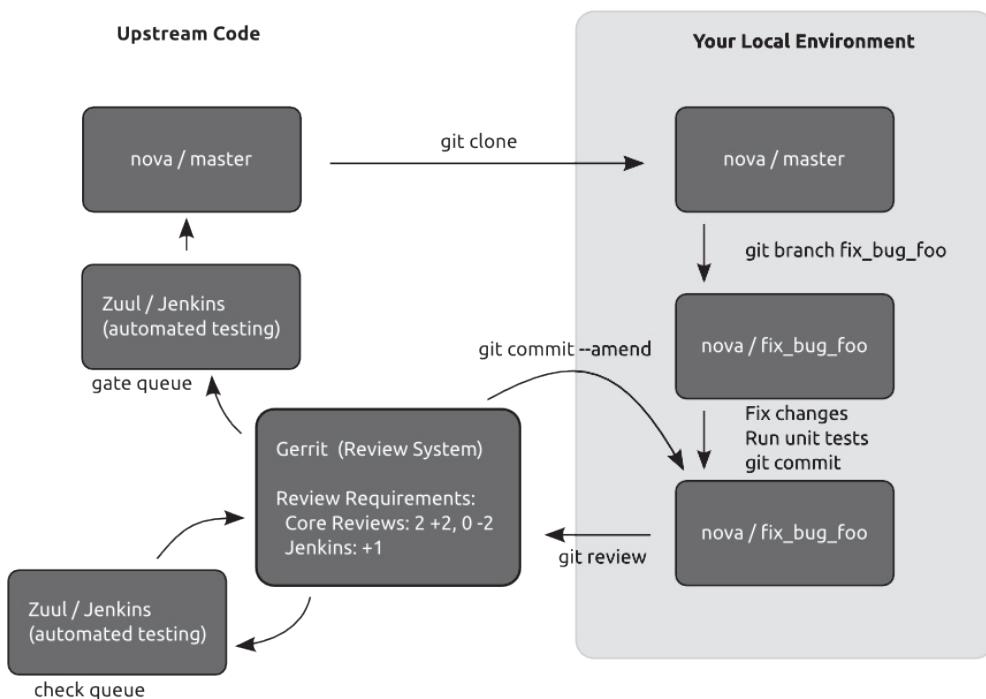
差分管理には Git を使っており、リポジトリは OpenStack Foundation のインフラで管理されています。Github にもリポジトリがありますが、こちらはあくまでミラーという扱いとなっており、プルリクエストを送ってもマージされません。Issue の登録もできませんので注意して下さい。

開発者

本家の集計によれば、世界 177ヶ国からのべ 32326人の開発者がいるそうです。開発者のバックグラウンドはさまざまです。企業に勤めている人もいれば、個人の趣味の一環で開発に貢献している人もいます。企業と言ってもこちらもまたさまざままで、ソフトウェア製品開発企業、社内のインフラで OpenStack を使っている企業、自社製品のドライバやプラグインを作成している企業などなどです。

コミュニティー

OpenStack WiKi に掲載されているレビューとマージのフロー図。
"Upstream Code" から git clone してくる部分から、右回りにフローを見ていくとわかりやすい。Zuul というのは CI 監視システムです。現在のテストキューの状態を確認することができます。





Individual Member: Free

There are two levels of accounts on OpenStack.org: "Community Member" and "Foundation Member". The Foundation Member level has additional rights but also substantial responsibilities as a member of the OpenStack Foundation. Here's a quick summary of the differences:

Which type of account would you like to create?

Community Member

- Submit Proposed Summit Talks - Allowed
- Vote on Proposed Summit Talks - Allowed
- Have a Profile on OpenStack.org - Included
- Vote on Foundation Board Member Elections - No

Community Member

Foundation Member

- Submit Proposed Summit Talks - Allowed
- Vote on Proposed Summit Talks - Allowed
- Have a Profile on OpenStack.org - Included
- Vote on Foundation Board Member Elections - Required*

Foundation Member

*Foundation Members are required to vote regularly in Board elections, or their membership can be suspended. Note that new Foundation Members must be members for 180 days before they are eligible to vote in Board Member elections. Inactive Foundation Member accounts are moved to Community Member level automatically after a period of inactivity. You may also log into your profile at any time and adjust your membership level.

OpenStack の開発に参加するには、まず OpenStack Foundation に参加する必要がある。個人では二通りの参加方法があるが、開発するために右の Foundation Member として登録すること。

Linux カーネルの開発コミュニティには「Linus Torvalds」という職業がありますが、OpenStack にはそのような役割の人はいません。開発を主導したり、意志決定を行うメンバーは開発者の選挙によって選ばれます。また、方向性の決定はそのままのメンバーを中心とし、開発者全体で議論されます。コミュニケーションツールとしては主にメーリングリストと IRC が使われています。

バグトラッキング

Launchpad というサービスを使っています。ちょっと重いのが難点……。Github の Issue 機能は使用されていませんので注意して下さい。Launchpad にはソフトウェア開発に便利な様々な機能があるのですが、その中でも Bug をバグトラッキングに、Blueprint を新規機能の開発管理に使っています。OpenStack コミュニティが Git リポジトリを持っていますので、Launchpad のコードリポジトリは使われていません。

コードレビュー

Gerrit というレビュートールを使って、マージ前のコードレビューと自動テスト、その後のメインツリーへのマージまで行っています。メーリングリストでのコードレビューは行われていません。また、Github のプルリクエストは受け付けていませんので注意して下さい。Gerrit に Push されたパッチは、ここで各パッチごとにコードレビューと自動テストを受けることになります。テストは Push と一緒にキューに入り、その結果はレビュー画面に一覧表示されます。

マージ

パッチがマージされるためには、自動テストをすべて Pass し、レビューからの指摘やコメントに適切に対応し、「コアレビュアー」と言われるプロジェクトの中心メンバー 2 人に "+2" というフラグを付けてもらう必要があります。2 つの "+2" フラグをもらい、コアレビュアーが "Workflow+1" というフラグを立てると、自動的に結合テストのキューに入ります。その結合テストに Pass すると、めでたく Master ブランチにマージされるという仕組みです。

仕様決定

新規機能の追加に関しては Launchpad の Blueprint という機能を使っています。バグチケットと同じように新機能の実装前に Blueprint チケットというものを作成します。このチケットで実装状況をトラッキングしていきます。以前はこのチケット上ですべてをまかなっていたのですが、最近では技術的仕様を SPEC という仕組みで議論することが多くなっています。と言っても、単に Blueprint が仕様を書くには手狭だった、というだけの話です。テキスト形式にしてコンポーネントごとに決められたフォーマットとテンプレートを使って仕様を書いて、それをコードレビューと同じ仕組みに乗せれば皆で議論できるし、その経過も残せるよね、というわけです。ほとんどの場合 Spec は ReST (ReStructuredText) 型式で書かれ、Gerrit 上でコメントをやりとりすることで進められています。



OpenStack 開発のための準備

メールアドレスを1つ作成する

これから OpenStack の開発に参加していくにあたり、まずメインとなるメールアドレスを決めましょう。このあと OpenStack 開発に必要なツールのアカウントでは、すべて同じメールアドレスを使用する必要があります。これはハマリポイントの1つですので注意しましょう。

最近は個人で多くのメールアドレスを持っていることが多いと思います。学校で配付されるものや、会社で使っているもの、個人用には Gmail、(まだあるのかわからないが、かつて「メッセ」と言われたものに使っていた) Hotmail など。

OpenStack に限らず、オープンソース開発ではメールアドレスは個人を特定するために重要で、公開が原則となっています。企業としてオープンソースコミュニティーに貢献していく場合は会社のドメインを持つメールアドレスを使うことが多いでしょう。OpenStack コミュニティーでも企業ドメインで活動している開発者も非常に多いです。貢献量を気にする場合は、Stackalytics というサービスで、個人のメールアドレスと所属組織を紐付けることができるので、個人の Gmail アドレスなどを使っても、あとからトラッキングすることができます。

私用のメールアドレスしか持っていない、あるいは会社のメールは転職時に使えなくなってしまうことを心配する場合は、これを機にオープンソース活動用のメールアドレスを1つ取得してみるのはどうでしょうか。今後も何かしらのオープンソース活動をする際にもとても便利だと思うので、オススメです。

OpenStack Foundation に参加する

すべての OpenStack 開発者は OpenStack Foundation へ参加することが必要です。まずは OpenStack Foundation にアカウントを作成します。このアカウント作成が、Foundation へ参加することになります。

<https://www.openstack.org/join/>

OpenStack Foundation メンバーには Community Member と Foundation Member の2種類あります。両者の違いはコミュニティーの舵取り役 (Board Member) の選ぶ際の選挙権があるか否かです。が、実は実際にコードを書き、パッチがマージされるためには Foundation Member であることが必要です。従って、ここでは Foundation Member を選んで、アカウントを作成します。アカウント作成時に氏名やメールアドレスなどを入力するのですが、分かりにくいと思われる箇所があるので少し解説します。

Affiliation

フォームのすぐ下に記載されている説明によると「過去 12 ヶ月間において、\$60,000 USD の給与契約を結んでいる組織」を書きます。要するに、勤め人の場合は会社名になりますね。あてはまる組織がない場合は、「Unaffiliated」と書けばよいみたいです。いやしかし、基準が超高給過ぎやしませんかね……。

Statement of Interest

日本語で言うところの「所信表明」です。決意表明とでも言いましょうか。何かカッコいいことを書いておきましょう。「Make the OpenStack world better」とか……。英語でカッコいいこと言うの難しいですね。製造業のテレビ CM のキャッチコピーみたいなのが参考にするといいかも？しれません？（適当）

Launchpad にサインアップする

OpenStack はチケットトラッカーに Launchpad というサービスを使っています。このアカウントも作成しておきましょう。

<https://launchpad.net/>





The screenshot shows the OpenStack Cloud Software settings interface. On the left, there's a sidebar with links like Profile, Preferences, Watched Projects, Contact Information, SSH Public Keys, HTTP Password, Identities, Groups, and Agreements. The main area has tabs for All, My, Projects, People, Documentation, Changes, Drafts, Draft Comments, Watched Changes, and Starred Changes. Under the 'Contact Information' tab, it says 'The following offline contact information is stored encrypted.' and provides fields for Mailing Address, Country, Phone Number, and Fax Number. A note states: 'Contact information will only be made available to administrators if it is necessary to reach you through non-email based communication. Received data is stored encrypted with a strong public/private key pair algorithm, and this site does not have the private key. Once saved, you will be unable to retrieve previously stored contact details.' Below this, it says 'Contact information last updated on May 11, 2014 at 5:33 AM.' and has a 'Save Changes' button.

Gerrit 上で ICLA にサインした状態

Contact Information が正常に保存された状態。個人情報保護の観点から、保存された値は画面からは見えず、最終更新日だけが分かるようになっている。

Gerrit にサインアップする

OpenStack はコードレビューに Gerrit というシステムを使っています。右上の Login をクリックしてログインします。このアカウントは Launchpad OpenID でのシングルレジストレーションになっています。

Gerrit の初期設定

Gerrit に少し初期設定が必要です。

ユーザー名

初回ログイン時にユーザー名を設定します。あとで変更はできないので気を付けましょう。

<https://review.openstack.org/#/settings>

Contact Information

<https://review.openstack.org/#/settings/contact>

これらを設定しないと、パッチを Gerrit に Push することができません。名前と Email アドレス、そして住所を設定します。Mailing Address というのは Email アドレスではなく、住所です。情報は暗号化されて保存されるので安心してください。とは言うものの、詳細な住所を書く必要は特になく、というのが本当のところです。市町村程度までで良いのではないかと思っています。法的には問題ありでしょうが……。電話番号もしかりです。登録に成功すると、

Contact information last updated on .

のように表示されます。

ICLA にサインする

<https://review.openstack.org/#/settings/new-agreement>

最後に Individual Contributor Licence Agreement への同意が必要です。ICLA を選んで設定しましょう。

git-review のインストール

git-review というのは、Gerrit にコードを Push する際に使う git コマンドのプラグインです。主に Python で作られています。



Chapter 2. パッチの作成

ます。インストール方法は、ディストリビューションのリポジトリを使う際は yum や apt-get などで、あるいは pip でのインストールもできます。

以上がパッチを書く前準備になります。結構いろいろありますね……。

バグの登録

バグチケットはまず Open 状態になります。その後、再現性があれば Confirmed、優先順位が付いたものは Triaged となり、開発者が修正を始めると In progress となります。修正が終われば最後は Close されるという一般的なライフサイクルをたどります。バグの優先順位付けやマイルストーンの決定は Triager と呼ばれるメンバーだけができるようになっています。とはいっても、見落としや間違い（Critical なバグの優先順位が低すぎるなど）もありますので、その場合はバグチケットにコメントを書くなどして、適切な順位付けを手伝ってあげて下さい。

これ以外のステータスとしては、ユーザーの設定ミスなどによる非バグの場合は Invalid、情報不足の場合は Incomplete などがあります。これらのバグに関しても、同じバグに遭遇したり情報を持っている場合は、コメントなどで助けてあげるといいと思います。「こっちの環境でも再現した！困っている！」と書くだけでも、Triager はだいぶ楽になるものです。

さて、パッチを書こうと言っても「何を直せばいいのかわからない」・「バグがない」といった声をよく聞きます。が、これは全くその通りです。少し前はバグが多かった（？）ので、ちょっと試してみるだけでもバグに遭遇しましたが、最近ではバグも減り、なかなかそういうこともなくなってきた。

そこで、まずはパッチを書いてみるにあたり、何のパッチを書くかについて考えてみました。まとめてみると、以下のような観点からバグを探すのがよいと思います。

使ってみて、不具合にぶち当たり、それを直す

これは結構ちゃんと使い始めないとなかなか無いかもしれません。手元のマシンで試しに動かしているだけという場合、開発者も同じような環境で開発していることが多いので、ちょっとしたバグには当たりにくいかかもしれません。

Launchpad にバグ登録されている中からみつける

すべての既知のバグは Launchpad に登録されている（はずです）ので、このなかから簡単そうなものを見つくりて直してみる方法です。とはいっても Nova のような大きなプロジェクトでは（残念というか選び放題というか）膨大な量のバグが登録されています。なので、このリストをながめてもなかなか見つかりません。こつとしては、

- Assignee が None になっているもの
- law-hanging-fluit タグが付いているもの
- といったものだけを検索してみるといいと思います。

コメントやドキュメントの修正を行う

まあ確かにソースコード内のコメントのタイプ修正だけというのは物足りないとは思いますが、どうもドキュメントの修正も同じように残念がる人が多くていけません。ドキュメントの不足や、自分が試してみた時に困った点や、そもそもドキュメントが現状と一致していない点を発見したら、どんどん直していきましょう。全くもって恥ずべきことではありません。OpenStack はプロジェクトが大きいぶん、ドキュメントの不足が目立ちます。「こんなドキュメントじゃわかんねー





```
~/sandbox $ git review --verbose --setup
2015-11-25 15:55:12.861166 Running: git config --get gitreview.scheme
2015-11-25 15:55:12.866903 Running: git config --get gitreview.hostname
2015-11-25 15:55:12.872724 Running: git config --get gitreview.port
2015-11-25 15:55:12.879181 Running: git config --get gitreview.project
2015-11-25 15:55:12.884850 Running: git log --color=never --oneline HEAD^1..HEAD
2015-11-25 15:55:12.891940 Running: git remote
2015-11-25 15:55:12.900731 Running: git config --get gitreview.username
No remote set, testing ssh://gerrit-username@review.openstack.org:29418/openstack-dev/sandbox.git
2015-11-25 15:55:12.907010 Running: git push --dry-run ssh://gerrit-username@review.openstack.org:29418/openstack-
dev/sandbox.git --all
ssh://gerrit-username@review.openstack.org:29418/openstack-dev/sandbox.git worked. Description: Everything up-to-date
Creating a git remote called "gerrit" that maps to:
    ssh://gerrit-username@review.openstack.org:29418/openstack-dev/sandbox.git
2015-11-25 15:55:16.316277 Running: git remote add -f gerrit ssh://gerrit-username@review.openstack.org:29418/openstack-
dev/sandbox.git
2015-11-25 15:55:24.065442 Running: git rev-parse --show-toplevel --git-dir
2015-11-25 15:55:24.071190 Running: git config --get remote.gerrit.pushurl
2015-11-25 15:55:24.078870 Running: git config --get remote.gerrit.url
2015-11-25 15:55:24.084521 Running: git config --list
Found origin Push URL: ssh://gerrit-username@review.openstack.org:29418/openstack-dev/sandbox.git
Fetching commit hook from: scp://gerrit-username@review.openstack.org:29418/hooks/commit-msg
2015-11-25 15:55:24.090996 Running: scp -P29418 gerrit-username@review.openstack.org:hooks/commit-msg .git/hooks/commit-msg
```

git-review の初期設定時のメッセージ。--verbose オプションをつけると Gerrit サーバーとの通信の様子を見ることができるが、SCP で hook スクリプトをダウンロードしてきていることが分かる。

よ！ハマったよ！」ということがありましたら、どんどん改善していくって欲しいと思います。

どの方法をとるにせよ、「直すのが簡単そうなバグ」というのはバグチケットからでは分かりにくいものです。ぱっとみ簡単そうに見えて、修正を始めたら実はとんでもない泥沼が……、なんてこともあります。特に、ぱっとみ簡単そうなのに、1年前から Assignee がとつかえひつかえ変わっているようなバグチケットを見つけたら、それは危険かもしれません。ちなみに、そのようなバグであっても自分が困っているのであれば、気合いを入れて直すべき、というのが本来の姿ではありますけどね。筆者の知り合いには、毎日の出勤電車の中でバグリストをながめて良さげなものを探すのを日課しているような人もいます。物件探しみたいです。

良い（？）を見つけることができたら、自分に Assign してしまいましょう。Unassined と書かれたカラムの隣に鉛筆マークがあります。ここから Assignee を変えることができます。あとで諦めて Assign を外すこともできますので、背水の陣の気持ちになる必要はありません。折角 OpenStack に貢献できるチャンスを、みすみす逃さないための予防措置のようなものだと思って下さい。

バグの新規登録

運良く（？）バグに遭遇した場合は、Lanchpad にバグ登録をしてみましょう。登録は Bug ページの右上にバグ登録リンク（Report a bug）があるのですぐできます。が、既に誰かがバグ登録をしているかもしれません¹⁾。Summary を入力して Next を押すと、既に似たバグが登録されていないか、Summary から推測して提示してくれます。もし、同じバグが既に登録されていた場合は、バグレポートが重複してしまいますので新規登録は避けましょう。そのかわり、そのバグにまだ Assignee がない場合は自分で修正できるチャンスですので、自身を Assign しちゃいましょう。

バグレポートの書き方

「バグ登録をするときにどう書けばいいのか分からない」と思う人が多いと思います。これ系の話は Web のいたるところに転がっている話ではあるのですが、少なくとも以下の情報は書くようにしましょう。

1. 怒りを静める

何よりも、まずこれが重要です。思ったより動かすのが大変でいらいらするのは分かりますが、それをバグレポートにぶつけるのはやめましょう。バグリストを見ていると、怒りにまかせた激しい文章をバグレポートにぶつけている例が散見されますが、これを反面教師としましょう。まずは怒りを静めること、これが大切です。

¹⁾ 大抵の場合、バグに遭遇した人は苛ついてますので、怒りにまかせてどんどんバグ登録をしてきます。割と既にバグ登録されていることが多いです。



2. 「どういう結果を期待しているのか」を自問する

これが意外と忘れられることがあります。バグというのは、動かないことではなく、期待した通りの動作をしないことです。「動かない！」(it doesn't work!)とか「さっきまで動いていたのに、何もしてないのに落ちた！(goes down!)」とだけ書くのではなく、自身がそのプロダクトに何を期待し、どのような結果を予想していたか、をまずメモに起こすところまで考えてみて下さい。このメモは次の実際のバグレポートを書く際に使います。

3. バグレポートを書く

ようやくバグレポートを書く段階ですね。バグレポートにあるとよい情報は、少なくとも以下です。

- ・バージョン
- ・やったこと
- ・それに帯する期待する動作
- ・実際の挙動
- ・なぜその挙動では困るのか
- ・なぜそのバグが重要なのか

こんなものだけでOKです。箇条書きでも問題ありません。特に「なぜその挙動では困るのか」と「どのぐらい重要なのか」を書くことがとても大切です。開発者は常に十分ではないため、99%のバグは緊急性が無いと判断されます。従って、そのなかでも「割と重要なんだよ」アピールをしておくと、優先度を付けやすくなります。バグの優先順位を付けるひとは、できるだけ重要度を下げようとしますので。

git-review の初期設定

とにかく、いざ修正を始める段階にきたとしましょう。リポジトリフォルダ内で、最初に git-review の初期設定をしておきましょう。

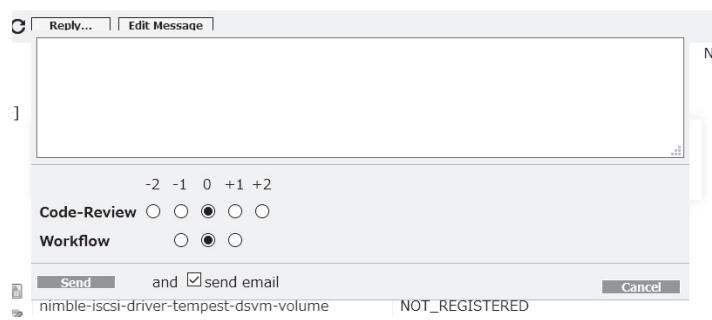
```
git config user.name 'name'  
git config user.email 'name@domain.com'  
git config gitreview.username 'gerrit username'  
git review --debug --setup
```

ここで Gerrit サーバーに SCP することで git-review の初期設定が行われます。具体的には、Gerrit サーバーが gerrit という名前でリモートに追加され、コミット時の hook スクリプトが設定されます。この hook スクリプトをダウンロードする際に、かなりのハイポート(29418)を使う仕様になっているので、Proxy 配下から接続する場合には接続エラーが発生するかもしれません。その場合はセットアップ前に以下の設定をしておくと、SSH ではなく HTTPS での接続で初期設定が行われます。

```
git config gitreview.scheme https  
git config gitreview.port 443
```

これらの設定値が有効になるためには、バージョン 1.25 以上の git-review である必要があります。pip ではこのバージョン

コアレビュアーになると "+2" フラグを操作できるようになる。Workflow フラグは CI キューに入るかのフラグです。"0" フラグは通常の CI キューでのテストに、"+1" フラグはコアレビュアーによるパッチ承認後の結合テストに進行することを意味します。また、"-1" フラグはどのテストキューにも入らないことを意味しますが、これはパッチを「下書き」状態にしたい時に便利です。





Zuul Status

Zuul is a pipeline oriented project gating and automation system. Each of the sections below is a separate pipeline configured to automate some portion of the testing or operation of the OpenStack project. For more information, please see the Zuul reference manual.

Queue lengths: 0 events, 0 results. Filters: Expand by default:

check	gate	post
(37)	(5)	(3)
Newly uploaded patches enter this pipeline to receive an initial +1 Verified vote from Jenkins.	Changes that have been approved by core developers are enqueued in order in this pipeline, and if they pass tests in Jenkins, will be merged.	This pipeline runs jobs that operate after each change is merged.
Change queue openstack/neutron ● openstack/neutron 2488510 0 min	Change queue openstack-infra/project-config ● openstack-infra/project-config 2489101 unknown 0 min	Change queue openstack/python-virageclient ● openstack/python-virageclient d4f75a8 unknown 57 min
Change queue openstack/neutron ● openstack/neutron 2488942 1 hr 12 min	Change queue openstack-infra/mistral ● openstack-infra/mistral 2493454 3 min	Change queue openstack-dev/devstack ● openstack-dev/devstack 8c5a31 unknown 56 min
Change queue integrated ● openstack-dev/devstack 2483488 13 min	Change queue openstack/openstack-manuals ● openstack/openstack-manuals 2491242 0 min	Change queue openstack/openstack-manuals ● openstack/openstack-manuals 9cc0dc9 unknown 49 min
Change queue openstack/magnum ● openstack/magnum 2489241 0 min	Change queue openstack/openstack-manuals ● openstack/openstack-manuals 249821 12 min	Change queue openstack/openstack-manuals ● openstack/openstack-manuals 9cc0dc9 tag 8 min

CIステータスを確認できるZuulというシステム。コードリース前になると、キューに膨大な量のテスト待ちが発生し、サイト全体が重くなったりする。コードリース前はコアレビューも多忙なため、そういう時期をさけてバッチを書くとレビューされやすい、という小先テクニックもある。

ンがインストールされますが、ディストリビューションによってはもっと古いバージョンの場合があります。1.25がインストールされなかつた場合は、

```
git remote add gerrit https://<gerrit.username>@review.openstack.org/
と自分でgerrit用のリモートを追加してから初期設定しましょう。
```

HTTPSでのセットアップをした場合、GerritへコードをPushするにはHTTPS用のパスワードを生成する必要がありますので、ここで生成しておきましょう。

```
https://review.openstack.org/#/settings/http-password
```

任意の文字列を設定することはできません。パスワードを設定しなおすことはできます。

ブランチの作成

まずバグ修正用のブランチを作成し、そのブランチをチェックアウトしましょう。

```
git checkout master
git branch bug/123456
git checkout bug/123456
```

bug/の後ろの数字はLaunchpadのバグ番号を書いておくとよいでしょう。と言っても、これはあくまで慣習であり、必ずこうでなければならない、という決まりはありません。が、慣習を破るとGerrit上でブランチ名が重複したりしますので、従っておくのが無難かと思います。

テスト

```
./run_tests.sh
tox. -e test
```

少なくとも一回はテストをしておきましょう。

コミットメッセージ

表面的には、最初の1行目はSummaryを書きます。ここは50文字以内にするのが慣習です。その後空行を一行あけ、コミットメッセージを書きます。こちらは79文字あたりで改行しましょう。

コミットメッセージに何を書くべきかというのも諸説あるところですが、少なくとも入れるべき項目は「なぜその変更をしたか」です。これだけはGitでトラッキングできないからです。バグ修正の場合は「～というバグを修正するため」と書いてよいですし、「～というバグを修正しないと～という影響が大きいため」という感じで書くとなお良いでしょう。反面教師もありますが、他の人のコミットメッセージを見てマネをするのもよいでしょう。



バグの詳細な説明は Launchpad 上で記述するため、コミットメッセージに書く必要はありません。かわりに、Launchpad のバグ ID をコミットメッセージに追記しておく必要があります。「Closes-Bug: #123456」といった書き方が慣習です。コミットメッセージの最後で、さらに空行を 1 行あけ、Launchpad のバグ番号を書いておきます。これをしておくと、自動的に Gerrit と Launchpad が連携され、パッチがコードレビュー中であることや、マージされたことなどのイベントが、Launchpad のコメントにも残るようになります。また、コミットメッセージでは詳細なバグの説明は書かないで、そのバグについての詳細を調べたい場合にも便利です。1 つのバグを直す際に複数の個別パッチが必要になった場合は「Related-Bug#123456」といった書き方や、Blueprint の場合は「Implement:.....」といった書き方をするときもあります。

適切なコミットメッセージを書くのはとても難しいものです。母語話者でもないですし、下手をすると実際にコードを書いている時間よりもコミットメッセージを考えている時間の方が長い時もあります²⁾。しかし、やはりコミットログは開発の履歴なので、正確に必要十分に書く必要があります。誰もが読むことができ、読んだ人が理解できるものであるべきなのです。

コミットメッセージが書けたら Commit します。この Commit 時に、自動的にコミットメッセージの最下行に「Change-Id:.....」というコメントが追記されます。Gerrit の初期設定時にダウンロードされた hook スクリプトがここで活きています。この ID は Gerrit がパッチを識別するのに使われています。これが変わってしまうと、別のパッチと扱われてしまうので注意が必要です。後述しますが、パッチを修正し再度レビューしてもらう際には、git commit に --amend オプションをつけ、前回のコミットを上書きして Gerrit に Push します。--amend 時に Change-Id を書き換えないように注意しましょう。

Gerrit にパッチを Push

コミットメッセージに Change-Id が追記されていることを確認したら、Gerrit にパッチを Push してみましょう。作業ブランチのまま、以下のコマンドを使います。

```
git review
```

正常に Push されれば、<https://review.openstack.org/#/c/.....> という URL で自分のパッチがレビューされるリンクが表示されます。あとはレビューされるのを待つだけになります。

レビュー

Push されたパッチはコアレビューの人人が順番に優先順位の高いものからレビューをしていきます。ですので、パッチをアップロードしてすぐに反応があることはあまりありません。パッチの数は膨大ですし、もちろん時差もあります。大抵の場合、Push して一発でパッチがマージされることはありません。直して欲しい点や疑問点などをコメントされますので、修正して再度アップロードすることになります。

ここで問題となるのは "-1" フラグです。先ほど、コードがマージされるまでにはコアレビューの人 2 名から "+2" フラグをもらう必要があると言いましたが、「このパッチには問題があるから、マージしないほうがいいかも」と思った場合、"-1" というフラグを付けられることがあります。この "-1" がフラグがあるとコードがマージされにくくなります。コアレビューから見ると、「まだ何か問題があるのかも?」と思われるがちだからです。

が、この "-1"、別にコードに対してイマイチな点があると言っているだけで、それほど凹むことではないことに気付けて下さい。別にパッチを送ったことに対して怒っているわけでも、ましてや人格を否定しているわけでもありません。大抵の場合は、ちゃんと理由があり、しかもそれなりにまっとうな理由（例えば使うべきライブラリを使用していないとか、スレッドセーフではないとか、環境に深く依存するコードになってしまっているとか）で "-1" フラグを付けています³⁾。

ただ、軽い気持ちで "-1" フラグを付けてくる人も結構います。これは OpenStack コミュニティーでも問題視されています。例えばタイプで "-1" フラグを付けたりとか、実装への質問で "-1" フラグを付けたりとか、最も困るのはコメント無しで "-1" フラグだけ付けてくる場合です。このような開発スピードをただ遅くするだけの "-1" フラグの濫用はやめよう、という動きになっています⁴⁾。

²⁾ なかなかうまい言い回しが思いつかずいろいろしてきて「これはエンジニアの仕事ではない」とか「建設的ではない」とか「なぜ変更したのか? って? とにかく俺の名前をコミットログに残したいからってだけだよ」とか言いたくなる気持ちはぐっと抑えましょう。そんなことを書いてもマージされないだけです。

³⁾ そもそもコメントさえもらはずに忘れ去っていくパッチがほとんどの中、コメントくれるだけでもラッキーと思うべきでしょう。

⁴⁾ まあこれは筆者の予想ですが、OpenStack 開発を仕事としてやっている人達はおおくいて（今の Linux カーネル開発も同様ですね）、そのような人達はフラグを付けた数を会社にトラッキングされていることがあります。つまり、建設的にコメントだけしてもメシの種にならんというわけです。まあ諸説ありますが。





```
~/sandbox $ git review
You are about to submit multiple commits. This is expected if you are
submitting a commit that is dependent on one or more in-review
commits. Otherwise you should consider squashing your changes into one
commit before submitting.
```

The outstanding commits are:

```
abcdefgh (HEAD -> master) Test commit 2
hijklmn Test commit 1

Do you really want to submit the above commits?
Type 'yes' to confirm, other to cancel: yes
remote: Resolving deltas: 100% (4/4)
remote: Processing changes: new: 2, refs: 2, done
remote:
remote: New Changes:
remote: https://review.openstack.org/123456
remote: https://review.openstack.org/789012
remote:
To ssh://gerrit-username@review.openstack.org:29418/openstack-dev/sandbox.git
 * [new branch]          HEAD -> refs/publish/master/bug/789012
```

A screenshot of the Gerrit web interface. At the top, there are buttons for 'Revisions ▾' and 'Download ▾'. Below this, a table lists two commits:

	Related Changes
:08	Test commit 2
:08	▶ Test commit 1

2つコミットオブジェクトを作成し、Gerrit にアップロードした場合のメッセージ例。レビュー URL が 2 つ作成されている。これらのパッチは依存関係があるとされ、Gerrit 上から自動的に依存のあるパッチへのリンクが作成される。初期設定時に HTTPS での接続をセットアップした場合は、途中で HTTPS のパスワードの入力を求められる。Gerrit の個人設定からあらかじめパスワードを生成しておこう。

レビュー後の修正

修正はやはりバグ修正ブランチでの作業になります。修正したらコミットするのですが、ここで注意して欲しいのは「新しいコミットオブジェクトを作成しない」ということです。具体的には

```
git commit --amend
```

のように、`amend` オプションで前回のコミットを上書きします。これには理由があります。先ほど最初にコミットを下さいに、`Change-Id` が生成されていることを確認しました。この `Change-Id` はコミットする際に自動的に生成され、コミットメッセージに追記されます。Gerrit 上ではこの `Change-Id` がパッチを識別する ID となっています。この ID が変わってしまうと、別のパッチとして Gerrit に扱われてしまうため、パッチの修正履歴やコメント履歴が無くなってしまいます。`--amend` オプションで ID を変えずに再度 `git review` でアップロードすることで、同じパッチの修正版としてアップロードすることができます。これで同じ URL にアクセスすることができますので、コメント履歴や修正履歴を残すことができます。レビューする側としても、前回パッチとの差分が簡単に把握できますので、レビューしやすくなります。

技術的な観点から言うと、Gerrit 上ではパッチをコミット ID ではなく独自の `Change-Id` で区別し、各 `Change-Id` のバージョンを `git` の `ref` オブジェクトに割り当てることで、「パッチのバージョン」という概念を持てるようにしています。別の見方をすると、コミット ID は違うけど、`Change-Id` は同じというわけです。

コードがマージされるまで、つまり問題点がなくなるまでこの修正と新バージョンのアップロードを繰り返すことになります。

マージ

コアレビュアー 2 名から "+2" フラグを付けられたらマージ対象となります。結合テストのキューに自動的に入り、終了後にマージされます。ほっとする瞬間です。



Chapter 3. パッチを書いたあと



バックポート

さて修正したバグですが、重要なバグと扱われた場合は、stable ブランチへのバックポートを依頼されることがあります。Launchpad のコメントから stable/kilo の方も修正してよついでに、なんて言われたりします。図らずも、2 つもコミットできちゃうというわけです。バックポートは、ローカルの作業としては git の cherrypick です。

```
git checkout master  
git pull
```

で最新のコードをダウンロードし、自身のパッチのコミット ID を見つけます（マージコミットの ID ではなく、マージ対象のコミット ID です）。たとえば kilo バージョンのブランチにバックポートする場合は、

```
git checkout stable/kilo
```

で kilo 用のブランチに移動し、

```
git cherrypick <CommitID>
```

とすることでバックポートできます。コンフリクトが起きなければ、あとは

```
git review stable/kilo
```

で kilo 用のブランチへのパッチとして Gerrit にアップロードします。その後のレビュー・プロセスは Master ブランチのそれと同じです。

コミットの分割

最初のうちはあまり大きなパッチを書くことはないですが、慣れてきてすこし大きなパッチ（新機能の開発など）を書くような場合にはコミットを分割してレビューを安全に行う必要が出てきます。あまり大きなコミットを作るとレビューしにくくなってしまうかもしれません。

パッチの分割は簡単で、つまり新しく commit するだけです。amend しない場合は新しいコミットオブジェクトが作成され、そのコミットには自動で新しい Change-Id が追記されます。Change-Id が違えば別のパッチとしてレビューにアップロードされますが、依存関係にある 1 つ前のパッチへのリンクが自動的にトラッキングされ、レビュー画面からも簡単にアクセスできるようになります。1 つ前のコミットを修正したら、その上有るコミットは rebase する必要があります。

コードレビューをする

ところで、コードレビューをする人ってどんな人だと思いますか？ 答えはあなたです。たぶん勘違いしていると思いますが、別にコアレビュアーでなくてもコードレビューをしてよいのです、というか、コアレビュアーではない開発者のレビューを大いに必要としています。「あれ？ こっちの実装の方がよくない？」といったコメントは大いに歓迎されます。「何でこんな実装をしているんだろう？」といった疑問は大いにあります。自分が分からることは他人も分かっていないことが多いです。

コアレビュアーは昔コアレビュアーではなかった人達です。割と熱心に他人のコードをレビューしコメントを書いているような人達が、コアレビュアーに推薦され "+2" フラグを付ける権限を受けます。何も特別な人達ではありません。

OpenStack に限らず、オープンソース開発においては誰もがコードレビューすることができます。

まずは自分が困っているバグの修正パッチをながめてみましょう。「ふーん。なんか動きそうだな。」って思ったら "+1"



フラグを付ければよし。「いや待て、その実装じゃうちじや動かん」と思ったらコメントすればよし。「何やってるかさっぱり分からんからコメント書くかわかり安いロジックに直してくれ」とか思ったら、その通りコメントすればよし。間違ったコメントしちゃったら「あ、そういうことか。ごめん勘違いした。+1」ってすればいいのです。試してみたいパッチがある場合、手元にダウンロードして動作を確認してみるとよいでしょう。git-review を使うと、他の人のパッチを簡単に手元の環境にダウンロードしてくることができます。

```
git review -l  
git review -d <Patch ID>
```

こまめにコードレビューをしていると、自分のパッチを気にかけてくれる人も増えますので自分のパッチがレビューされやすくなり、最終的にはマージされやすくなります。何か見たことある人のパッチが何となく優先度が上がります。なかなかウェットな世界で OpenStack は作られています。

Blueprint

OpenStack では活発に新規機能の追加や実装が行われています。新規機能の開発では主に Launchpad の Blueprint 機能が使われてきましたが、最近ではこれに加えて Spec のレビューということが行われています。

新規機能をマージしてもらうには、まず Blueprint の登録とコアレビューによる承認が必要です。まず、Blueprint を新規作成し、簡単な説明を書きます。その後、定期的に行われている IRC でのミーティングに参加し、そこで承認の依頼をし、多少の受け答えをし、妥当なものだと判断されると、その blueprint は承認されます。ここで初めて開発パッチを Gerrit にアップロードし、レビューを受けることができます。

さて新規機能の開発には、その必要性や仕様、目標バージョンなどを議論して決定しなければなりません。このように新規機能の開発に必要な Blueprint への情報やその書き方は、プロジェクトごとにテンプレートが決められていますので、Wiki などで探すとよいでしょう。小さなプロジェクトの場合はそのまま Blueprint で議論が進みます。Nova や Neutron のような大きなプロジェクトでは、Spec ファイルによる設計レビューが行われています。Spec ファイルのテンプレートも用意されていますが、要するに RST 型式のテキストファイルで、Blueprint だと狭くて書きにくい仕様の詳細を議論しているだけですので、Blueprint とあまり差は無いとも言えます。

Spec の議論では、必要となる技術要素や仕様はもちろん、新規機能の必要性や妥当性なども議論されます。また、代替案なども書く必要もあります。最初からこの仕様を書き上げるのは難しいので、バグの修正に飽きたり、自分も欲しいと思っていた機能などの Spec があれば、まずはそのレビューに参加してみましょう。OpenStack の勉強にもなります。

Gerrit を便利に使う

まあ正直言って、イマドキの他のレビューシステムと見比べると、Gerrit の見た目はしょっぱい感じなのは否めません。いや、まあ機能はかなり豊富なんですね。まあ見た目はブラウザのプラグインなどで変えられます。また、Setting を変更することでも少し変えることができます。

<https://review.openstack.org/#/settings/preferences>

画面が広い場合は、“Display Person Name In Review Category”にチェックを入れておくとよいと思います。というのも、これをオンにするとパッチのリストにパッチの承認をする人の名前が表示されるため、プロジェクトの主要人物を把握できるからです。先ほども述べたように、なかなかウェットな世界ですので。やはりまずは名前を把握することが大切です。営業の基本ですね。また Change view を New Screen にしたほうがよいでしょう。レビュー画面が少し観やすくなります。

まあいずれにしても若干しょっぱい感じはいなめません。

オープンソースソフトウェアをオープンソースコミュニティーで開発す



るということ、あるいは冷静と情熱のあいだにある忍耐について

おどかすわけではないですが、初めて書いたパッチがマージされるのは簡単ではありません。マージされない判断が降りればいい方で、大抵の場合は無視され（たように感じ）ます。パッチがマージされるまでどのぐらいかかるかというと、まちまちです。パッチを送ったその日にマージされることもあります、三ヶ月かかることもあります。まず、マージ以前にレビューされなという事件が起こります。無視されたりします。まあ故意ではないのですが、コアレビュアーと言われている人はそもそも本業が忙しいのです⁵⁾。

オープンソースコミュニティには実にいろいろな背景の人がいますが、主には趣味人と会社員です。趣味人は趣味で開発をしますが、会社員は企業としてオープンソースコミュニティに貢献しています。いずれにせよ、全く無関心にパッチを書いたりしているわけではありません。それぞれ何かしら利益があってやっています。個人は趣味の充実かもしれませんし、企業は広報活動の一環だったりするかもしれません。

とはいって、コアレビュアーの人がレビューをしてくれるのは単に忙しいからではなく、パッチを送る側にも問題がある場合もあります。1つは、慣習やルールに従っていない場合です。で、もう一つ重要なのは、コアレビュアーではない一般開発者のレビューやコメントをないがしろにしている場合です。コアレビュアーは自分の判断基準でパッチの善し悪しを把握しているわけではありません。他の人がどんな意見を出しているか、他の人がより良い意見を出していないか、バグを発見していないか、理解していない点はないか、などです⁶⁾。

で、故意ではないわけなので、「あ、なんか忘れられてるかも？」って思った時は何かしらの方法でレビューをお願いするってこともできます。ではどのぐらいレビューが付かないと不安になるべきかと言うと、1週間ぐらいが目安のようです。なぜ1週間かというと、プロジェクトによっては毎週レビューをみんなでやる日を決めている場合があり、その時に一気にコメントが付いたりします。また、コアレビュアー自身が気合いを入れてレビューをする日、みたいなを決めたりしている場合もあります。本業のスケジュールも鑑みて、週1しか時間が取れない人もいます。しかし1週間というのは、これまたやきもきしますね。

お願いは大抵IRCです。注意すべきは、メーリングリストにレビューの依頼を投げるのはルール違反ということです。レビューの依頼を投げてもいいのは、非常に重要なバグやもめそうな仕様決定、プロジェクト横断的なSPECレビューなどのみです。いらっしゃるメーリスに個人のレビュー依頼を投げると、「ここはそのような場ではありません。週次のIRCミーティングで言って下さい。」って注意されます。たまにそういうやりとりを見かけます。IRCでちらっとお願いして、また1週間待ってみる。そんなことを4回くらい繰り返しているうちに、なんとかなります。5回以上は避けましょう。

実は、やはりコミュニティとの関わりかたは個人の性格に依存してしまうものです。

オープンソース活動をするということ

OpenStackへの貢献方法を書いてきましたが、文字ばっかりで疲れてしましました。

オープンソースのソフトウェアを本番環境で使うことなんてあり得ない、と言う場合があります。諸説ありますが、大抵の場合サポートがない、つまり人のせいにできないからというのが理由です。非常に重要な観点であることは否定しませんが、そんなものお前が謝るかクビになればいい話なので、あまり説得力はありません。そういう人、意外と営業担当、つまり本当にお金を生み出す人達ともうまくいかないことが多い気がします。営業担当からしてみれば、トラブル時の責任の所在なんぞ甚だどうでもよいことであって、トラブル時の金銭的損失に対する謝罪を変えよ営業を舐めるなって思うんじゃないかと僕は思うんです。「オープンソースを導入すればオープンイノベーションを使えたり、先進機能を使えたり、標準的な技術仕様を展開できたりするかもしれませんし、ビジネスチャンスとその実装を近づけることができるかもしれません、サポートに責任を押しつけられないのでやりません僕のせいじゃないです仕方無いんで」、「ふざけるな勝手に責任者ズラするんじゃねえこの金食い虫が儲けの種を潰すな」ってなると思うんですが……。そんなことないのかな……。

まあいずれにせよ、ハズレOSSがある以上は何か選ぶ基準みたいなものは必要かもしれません。1つは、開発の活発さです。最終コミットが5年前とか、はちょっともう開発されてなさそうですね。プロジェクトの年齢も気になります。1日100コミットぐらいしていくて活発だけど、最初のコミットから3日しか経っていない、というのはちょっと品質が心配で

⁵⁾ 本業が忙しくないひとはコアレビュアーになれるほど人望も技術力もないのでしょうか。世の中は非対称です。

⁶⁾ 「俺はマージがされたいんだよ。コアだけ来ればいいんだよ。コアは俺がマージされたい気持ち分かれよ。」って思っていらっしゃる人はオープンソース開発全般に向いてないのでやめましょう。他の人の迷惑です。マージから最も遠いです。そういうことを分からずにマージの目標だけを早急に達成したがる上司は痛々しくて見てはいけないのでそっとしておきましょう。精神の毒です。



す。誰が作っているのか、というのも重要でしょう。良くも悪くも、信頼できる OSS というのはバックに大企業がついていることが多いです。主にバックの企業の従業員がコードを書き、一般的なユーザーがバグとりやテスト、仕様の議論をするというスタイルが一般的です。そのような観点からすると、OpenStack はもう何年も開発され、未だに活発で、バックに複数の企業がついているという三拍子そろった OSS になります。





Chapter 4. Object Storage の概要



Object Storage とは何か

オブジェクトストレージとは、データをオブジェクトとして管理するストレージシステムのことである。これは、データをファイル階層として扱うファイルシステムとも、データをセクターとトラックによるブロック構造として扱うブロックストレージとも異なるストレージ形態である。基本的にオブジェクトは、データそのものと、加えて様々なメタデータとユニークな ID 情報を持っている。通常、NAS や CIFS ストレージはディレクトリによる階層構造を持っているのに対して、オブジェクトストレージはフラットなオブジェクト構造を持っている。このオブジェクト構造により、オブジェクトの移動が容易となること、オブジェクト毎にメタデータを付与できること、ACL 設定が可能であること、オブジェクト数を原則として無制限に拡張可能（ディレクトリの inode サイズの制限を受けない）であること、ユーザ間での共有が容易であること、などの特徴をもつことができる。

オブジェクトストレージが適しているデータとその用途

オブジェクトストレージは、分散ストレージであるがゆえに一貫性の担保が非常に難しい。したがって、全ての用途に適用できるわけではない。適用できるデータの例としては、非構造データ、写真、動画、個人データ、学術データなどである。また、そのデータの用途としては、SNS を始めとしたソーシャルコンテンツ、静的な Web コンテンツ、クラウドストレージ、バックアップなどが挙げられる。

Swift の概要

Swift の特徴

Swift は、OpenStack のコアコンポーネントの中でオブジェクトストレージを担う部分のプロジェクトである。Keystone によるマルチテナント化されたアカウント構造と、S3 のバケットに相当するコンテナという、いわゆる NAS のボリュームにと同等の構造を持っている。オブジェクトの最大サイズは 5GB であり、データのストアには REST API を利用することができる。また、デフォルトで 3 冗長にデータが複製され保存される。

基本構成

Swift は、他の OpenStack プロジェクト同様 Python で開発されている。しかし、バージョン管理は他のコンポーネントとは異なり、独自のバージョン番号により管理されている。そのため、半年ごとの OpenStack のリリース時には、そのタイミングでの最新の Swift が採用されることになる。

ファイルシステム

ファイルシステムには、デフォルトで XFS が推奨されている。これは、object-server に xattr というファイルシステムの拡張属性が利用されているからである。よって、ext4 を用いて Swift を構築することも可能である。



基本構造

基本構造として、Proxy ノードと Storage ノードに分けることができる。

Proxy ノード

Proxy ノードは、フロントエンドのサーバとして REST API のリクエストを受け付ける。Storage ノードは、Proxy ノード上でコンシスティントハッシングにより、どのノードに配置するかが決定されたうえで、Storage ノード上に実データが配置される。

Storage ノード

Storage ノードは、内部では account-server、container-server、object-server の 3 つが実際に稼働している。account-server と container-server は SQLite が利用され、object-server は XFS のファイルシステムが利用されている。

account-server

account-server は、Keystone で管理されているアカウントごとの SQLite が作成され、その中で各アカウントごとのコンテナ情報が格納されている。

container-server

container-server は、各コンテナごとの実オブジェクトの各種情報が格納されている。各コンテナごとに 1 つの SQLite ファイルを持つ構造となっている。

object-server

object-server は、実オブジェクトが計算されたハッシュ値に基づき、ノード配置とディレクトリ構造が決定されたうえで、実際に XFS ファイルシステム上にストアされる。

Zone

Swift クラスターは、ゾーンという論理的なグルーピングがなされており、データのレプリケーションは基本的にこのゾーンをまたぐ形で行われる。

Ring ファイル

Storage ノードの分散配置の構造自体は Ring ファイルという静的なファイルを全てのノードに配布することで決定される。このファイルの情報に基づき、ノードの拡張、縮退が実施される。データ格納用ディスクは、RAID を組まず（あるいは Single Disk RAID0 を組む）、1 本ずつ Ring ファイルに登録する。つまり 1 つのディスクが 1 つのノードのように振る舞うことになる。

小規模構成で作る Swift クラスター (Liberty Release)

はじめに

OpenStack この項では、仮想環境上の VM を使って、小規模構成で Swift を構築してみる。ただ、Liberty Release 時点での公式ドキュメントだけでは構築はうまくいかない。構築出来ない原因は、container-reconciler が memcached を必要とすることが公式ドキュメントのどこにも記載されていなかったためと、root デバイスしかない仮想 VM 上で構築しようとしたためである。



Component	Num of VMs	IP address
Proxy	1	192.168.0.2
Account & Container	5	192.168.0.[3-7]
Object	5	192.169.0.[8-12]
Keystone & MySQL	1	192.168.0.13

memcached に関しては、以下のとおり issue が存在する。container-reconciler need memcache in storage node⁷⁾
この点を踏まえて、改めて構築の手順を以下に記すことにする。

構成

OS は Ubuntu 14.04.3 を利用し、かつ、Swift は Liberty リリース時点のものを用いた。加えて言うと、認証には同じく Kilo リリースの Keystone を使った。

account-server/container-server と object-server を別の VM 上に構築しているため以下の様な構成となっている。また、Region は 1 つ、Zone は 5 つで構築した。各 Zone に付き、account-server/container-server を 1VM と object-server を 1VM の計 2VM 使っている。

keystone の構築

Keystone 用 MySQL サーバの設定

MySQL サーバは構築済みであることを前提として、DB を作成する。

MySQL クライアントを利用して、DB サーバに接続する。

```
$ mysql -u root -p
```

keystone という名前で DB を作成する。

```
CREATE DATABASE keystone;
```

パスワード付きでユーザを作成し権限を与える。KEYSTONE_DBPASS は、任意のパスワードに置き換える。

```
GRANT ALL PRIVILEGES ON keystone.* TO 'keystone'@'localhost' \
    IDENTIFIED BY 'KEYSTONE_DBPASS';
GRANT ALL PRIVILEGES ON keystone.* TO 'keystone'@'%' \
    IDENTIFIED BY 'KEYSTONE_DBPASS';
```

Keystone インストール

Keystone は、Kilo リリース以降 WSGI server ではなく Apache2 で稼働するように変更になった。Ubuntu14.04 では、Upstart が init システムに使われているため、以下のファイルを /etc/init 配下に配置することで元となる Keystone の WSGI Server が自動起動しないようにする必要がある。

```
# echo "manual" > /etc/init/keystone.override
必要なパッケージをインストールする。
# apt-get install keystone apache2 libapache2-mod-wsgi memcached python-memcache
/etc/keystone/keystone.conf を記載していく。ADMIN_TOKEN に任意の文字列を入れる。この文字列が外部に漏れると OpenStack 環境を奪取されてしまうので決して漏らしてはならない。
```

```
[DEFAULT]
...
admin_token = ADMIN_TOKEN
```

DB の項目には以下のように必要な項目を記載する。controller には DB サーバのホスト名、もしくは IP アドレスを書く。

⁷⁾ <https://bugs.launchpad.net/openstack-manuals/+bug/1464939>



```
[database]
...
connection = mysql+pymysql://keystone:KEYSTONE_DBPASS@192.168.0.13/keystone
デフォルトの Keystone のユーザ認証用の Token のストア先が、Liberty 以降データベースから memcached に変更になった。よって、keystone.conf に以下の項目を記載する必要がある。memcache 項目の localhost には、memcached が稼働するホストのホスト名か IP アドレスを記述する。
[memcache]
...
servers = localhost:11211
...
[token]
...
provider = uuid
driver = memcache
...
[revoke]
...
driver = sql
ここまで編集した状態でデータベースのスキーマを作っていく。
# su -s /bin/sh -c "keystone-manage db_sync" keystone
次に、Apache の設定を行う。

/etc/apache2/apache2.conf に以下を追記する。controller には、Keystone の IP アドレスか、もし Keystone の前段にロードバランサを設置しているのであれば、ロードバランサの IP アドレスを記載する。
ServerName 192.168.0.13

/etc/apache2/sites-available/wsgi-keystone.conf に以下を記述する。なお、processes=5 としているが、ここは使用するサーバーに合わせて変更する。
Listen 5000
Listen 35357

<VirtualHost *:5000>
    WSGIDaemonProcess keystone-public processes=5 threads=1 user=keystone group=keystone
    display-name=%{GROUP}
    WSGIProcessGroup keystone-public
    WSGIScriptAlias / /usr/bin/keystone-wsgi-public
    WSGIApplicationGroup %{GLOBAL}
    WSGIPassAuthorization On
        <IfVersion >= 2.4>
            ErrorLogFormat "%{cu}t %M"
        </IfVersion>
    ErrorLog /var/log/apache2/keystone.log
    CustomLog /var/log/apache2/keystone_access.log combined

    <Directory /usr/bin>
        <IfVersion >= 2.4>
            Require all granted
        </IfVersion>
        <IfVersion < 2.4>
            Order allow,deny
            Allow from all
        </IfVersion>
    </Directory>
</VirtualHost>

<VirtualHost *:35357>
    WSGIDaemonProcess keystone-admin processes=5 threads=1 user=keystone group=keystone
    display-name=%{GROUP}
    WSGIProcessGroup keystone-admin
    WSGIScriptAlias / /usr/bin/keystone-wsgi-admin
    WSGIApplicationGroup %{GLOBAL}
    WSGIPassAuthorization On
        <IfVersion >= 2.4>
            ErrorLogFormat "%{cu}t %M"
        </IfVersion>
    ErrorLog /var/log/apache2/keystone.log
    CustomLog /var/log/apache2/keystone_access.log combined
```



```
<Directory /usr/bin>
  <IfVersion >= 2.4>
    Require all granted
  </IfVersion>
  <IfVersion < 2.4>
    Order allow,deny
    Allow from all
  </IfVersion>
</Directory>
</VirtualHost>
```

以下のとおり、シンボリックリンクを張る。

```
# ln -s /etc/apache2/sites-available/wsgi-keystone.conf /etc/apache2/sites-enabled
Apache2 を起動する。
# service apache2 restart
```

Service/Endpoint の作成

ここから先は、openstack コマンドを用いた操作となる。ADMIN_TOKEN には、keystone.conf に記した Token の文字列を入れる。controller には、Keystone のホスト名、IP アドレス、もしくは、ロードバランサを使っている場合、ロードバランサの IP アドレスなどを入れる。

```
$ export OS_TOKEN=ADMIN_TOKEN
$ export OS_URL=http://192.168.0.13:35357/v3
$ export OS_IDENTITY_API_VERSION=3

keystone サービスを作成する。

$ openstack service create \
  --name keystone --description "OpenStack Identity" identity
+-----+
| Field      | Value
+-----+
| description | OpenStack Identity
| enabled     | True
| id          | 4ddaae90388b4ebc9d252ec2252d8d10
| name        | keystone
| type        | identity
+-----+
```

Keystone v2.0 用の Endpoint を追加していく。controller には、先ほどと同じように Keystone の情報を入れる。

```
$ openstack endpoint create --region RegionOne \
  identity public http://192.168.0.13:5000/v2.0
+-----+
| Field      | Value
+-----+
| enabled     | True
| id          | 30fff543e7dc4b7d9a0fb13791b78bf4
| interface   | public
| region     | RegionOne
| region_id  | RegionOne
| service_id | 8c8c0927262a45ad9066cf70d46892c
| service_name| keystone
| service_type| identity
| url         | http://192.168.0.13:5000/v2.0
+-----+

$ openstack endpoint create --region RegionOne \
  identity internal http://192.168.0.13:5000/v2.0
+-----+
| Field      | Value
+-----+
| enabled     | True
| id          | 57cfa543e7dc4b712c0ab137911bc4fe
| interface   | internal
| region     | RegionOne
| region_id  | RegionOne
| service_id | 6f8de927262ac12f6066cf70d99ac51
| service_name| keystone
| service_type| identity
| url         | http://192.168.0.13:5000/v2.0
+-----+
```



```
$ openstack endpoint create --region RegionOne \
    identity admin http://192.168.0.13:35357/v2.0
+-----+-----+
| Field | Value |
+-----+-----+
| enabled | True |
| id | 78c3dfa3e7dc44c98ab1b1379122ecb1 |
| interface | admin |
| region | RegionOne |
| region_id | RegionOne |
| service_id | 34ab3d2726ac449cba6cf704dbc11f |
| service_name | keystone |
| service_type | identity |
| url | http://192.168.0.13:35357/v2.0 |
+-----+-----+
```

Project/User/Role の作成

Admin 用 Project を作成する。

```
$ openstack project create --domain default --description "Admin Project" admin
+-----+-----+
| Field | Value |
+-----+-----+
| description | Admin Project |
| domain_id | default |
| enabled | True |
| id | 343d245e850143a096806dfaef9afdc |
| is_domain | False |
| name | admin |
| parent_id | None |
+-----+-----+
```

admin ユーザの作成を行う。

```
$ openstack user create --domain default --password-prompt admin
User Password:
Repeat User Password:
+-----+-----+
| Field | Value |
+-----+-----+
| domain_id | default |
| enabled | True |
| id | ac3377633149401296f6c0d92d79dc16 |
| name | admin |
+-----+-----+
```

admin 用 role の作成を行う。

```
$ openstack role create admin
+-----+-----+
| Field | Value |
+-----+-----+
| id | cd2cb9a39e874ea69e5d4b896eb16128 |
| name | admin |
+-----+-----+
```

admin ユーザを admin ロールに紐付ける。

```
$ openstack role add --project admin --user admin admin
```

Service 用 Project を作成する。

```
$ openstack project create --domain default \
    --description "Service Project" service
+-----+-----+
| Field | Value |
+-----+-----+
| description | Service Project |
| domain_id | default |
| enabled | True |
| id | 894cdfa366d34e9d835d3de01e752262 |
| is_domain | False |
| name | service |
| parent_id | None |
+-----+-----+
```



demo 用 Project を作成する。demo プロジェクトは、一般ユーザとして swift の操作に使用する。

```
$ openstack project create --domain default \
  --description "Demo Project" demo
+-----+-----+
| Field | Value |
+-----+-----+
| description | Demo Project
| domain_id | default
| enabled | True
| id | ed0b60bf607743088218b0a533d5943f
| is_domain | False
| name | demo
| parent_id | None
+-----+-----+
```

demo ユーザを作成する。demo ユーザは、一般ユーザとして swift の操作に使用する。

```
$ openstack user create --domain default \
  --password-prompt demo
User Password:
Repeat User Password:
+-----+-----+
| Field | Value |
+-----+-----+
| domain_id | default
| enabled | True
| id | 58126687cbcc4888bfa9ab73a2256f27
| name | demo
+-----+-----+
```

user 用 role を作成する。

```
$ openstack role create user
+-----+-----+
| Field | Value |
+-----+-----+
| id | 997ce8d05fc143ac97d83fdfb5998552
| name | user
+-----+-----+
```

demo ユーザを user ロールに紐付ける。

```
$ openstack role add --project demo --user demo user
```

環境変数ファイルの作成

ここまで環境すると ADMIN_TOKEN ではなく、Keystone v3 API を使用した操作が可能となる。admin 用、demo 用で以下のようなファイルを作成する。

admin 用に admin-openrc.sh というファイルを作成し、以下を追記する。

```
export OS_PROJECT_DOMAIN_ID=default
export OS_USER_DOMAIN_ID=default
export OS_PROJECT_NAME=admin
export OS_TENANT_NAME=admin
export OS_USERNAME=admin
export OS_PASSWORD=ADMIN_PASS
export OS_AUTH_URL=http://192.168.0.13:35357/v3
export OS_IDENTITY_API_VERSION=3
```

demo 用に demo-openrc.sh というファイルを作成し、以下を追記する。

```
export OS_PROJECT_DOMAIN_ID=default
export OS_USER_DOMAIN_ID=default
export OS_PROJECT_NAME=demo
export OS_TENANT_NAME=demo
export OS_USERNAME=demo
export OS_PASSWORD=DEMO_PASS
export OS_AUTH_URL=http://192.168.0.13:5000/v3
export OS_IDENTITY_API_VERSION=3
```

Keystone における Swift ユーザーの作成 Endpoint の登録

Libery リリースを用いるため基本操作には v3 API と python-openstackclient を利用する。



Swift サービスの作成、Swift ユーザの作成、Swift ユーザーの admin ロールへの紐付けは、公式ドキュメントの通りである。

Swift サービスを作成する。

```
$ openstack service create --name swift \
    --description "OpenStack Object Storage" object-store
+-----+
| Field      | Value
+-----+
| description | OpenStack Object Storage
| enabled     | True
| id          | 75ef509da2c340499d454ae96a2c5c34
| name        | swift
| type        | object-store
+-----+
```

swift ユーザを作成する。

```
$ openstack user create --domain default --password-prompt swift
User Password:
Repeat User Password:
+-----+
| Field      | Value
+-----+
| domain_id | default
| enabled    | True
| id         | d535e5cbd2b74ac7bfb97db9cced3ed6
| name       | swift
+-----+
```

swift ユーザを admin ロールに紐付ける。

```
$ openstack role add --project service --user swift admin
```

API Endpoint の登録

```
$ openstack endpoint create --region RegionOne \
    object-store public http://192.168.0.2:8080/v1/AUTH_%(tenant_id)s
+-----+
| Field      | Value
+-----+
| enabled    | True
| id         | 12bfd36f26694c97813f665707114e0d
| interface   | public
| region     | RegionOne
| region_id  | RegionOne
| service_id | 75ef509da2c340499d454ae96a2c5c34
| service_name| swift
| service_type| object-store
| url        | http://192.168.0.2:8080/v1/AUTH_%(tenant_id)s
+-----+
```

```
$ openstack endpoint create --region RegionOne \
    object-store internal http://192.168.0.2:8080/v1/AUTH_%(tenant_id)s
+-----+
```

```
| Field      | Value
+-----+
| enabled    | True
| id         | 7a36bee6733a4b5590d74d3080ee6789
| interface   | internal
| region     | RegionOne
| region_id  | RegionOne
| service_id | 75ef509da2c340499d454ae96a2c5c34
| service_name| swift
| service_type| object-store
| url        | http://192.168.0.2:8080/v1/AUTH_%(tenant_id)s
+-----+
```

```
$ openstack endpoint create --region RegionOne \
    object-store admin http://192.168.0.2:8080/v1
+-----+
```

```
| Field      | Value
+-----+
| enabled    | True
| id         | ebb72cd6851d4defabc0b9d71cdca69b
| interface   | admin
+-----+
```



region	RegionOne
region_id	RegionOne
service_id	75ef509da2c340499d454ae96a2c5c34
service_name	swift
service_type	object-store
url	http://192.168.0.2:8080/v1

ここで注意しなければならない点は、Endpoint のアドレスである。公式ドキュメントでは controller と記載がある。これは Swift の proxy-server が controller で稼働していることが前提となっている。proxy-server を別サーバとして構築、あるいは、前段にロードバランサを用いる場合、この Endpoint に記載するアドレスは proxy-server、もしくは、ロードバランサの IP アドレスである。ポートは 8080 に設定されているが、これは 80 に変更しても問題ない。

上記では、proxy-server の IP address を用いて設定している。

proxy-server の設定

パッケージのインストール

```
# apt-get install swift swift-proxy python-swiftclient \
  python-keystoneclient python-keystonemiddleware \
  memcached
```

GitHub からサンプル conf を取得して編集を加える。

```
# curl -o /etc/swift/proxy-server.conf
https://git.openstack.org/cgit/openstack/swift/plain/etc/proxy-server.conf-
sample?h=stable/liberty
```

認証項目に関してのみ設定値に注意が必要である。

```
[DEFAULT]
...
bind_port = 8080
user = swift
swift_dir = /etc/swift
...
[pipeline:main]
pipeline = catch_errors gatekeeper healthcheck proxy-logging cache container_sync bulk
ratelimit authtoken keystoneauth container-quotas account-quotas slo dlo versioned_writes
proxy-logging proxy-server
...
[app:proxy-server]
use = egg:swift#proxy
...
account_autocreate = true
...
[filter:keystoneauth]
use = egg:swift#keystoneauth
...
operator_roles = admin,user
...
[filter:authtoken]
paste.filter_factory = keystonemiddleware.auth_token:filter_factory
...
auth_uri = http://192.168.0.13:5000
auth_url = http://192.168.0.13:35357
auth_plugin = password
project_domain_id = default
user_domain_id = default
project_name = service
username = swift
password = SWIFT_PASS
delay_auth_decision = true
...
[filter:cache]
use = egg:swift#memcache
...
memcache_servers = 127.0.0.1:11211
```

公式ドキュメントでは、auth_uri と auth_url に controller と記載があるが、この項目は認証に関する項目であるため、keystone が稼働しているサーバのアドレスに変更する必要がある。



account-server、container-server、object-server の下準備

Swift を構築するにあたって、account-server、container-server、object-server では root 領域とは別デバイスのパーティションが必要となる。例えば、object-server では object-replicator というプロセスがデータ領域としての追加デバイスがマウントされているかどうかのチェックをしている。root 領域しかない場合、プロセスの起動に失敗するため注意が必要である。

sdb 以降のデバイスを確保できる場合はそれを使って XFS でフォーマットする。無い場合は、ループバックデバイスを用いる方法がある。

ループバックデバイスを使って普通のファイルを XFS ファイルシステムとしてマウントする

ちなみに、以下の方法は devstack のソースコードをベースにしている。

XFS パッケージのインストール

xfsprogs というパッケージが必要になる。

```
$ sudo apt-get install -y xfsprogs
```

ループバックデバイスとなるファイルの作成

ファイルを配置するディレクトリの作成と、touch によるファイル作成を行う。

```
$ sudo mkdir -p /srv/node/images  
$ sudo touch /srv/node/images/file.img
```

truncate コマンドにより Sparse ファイルを作成する。

```
$ sudo truncate -s 50G /srv/node/images/file.img
```

ファイルシステムのフォーマット

XFS でフォーマットする

```
$ sudo /sbin/mkfs.xfs -f -i size=1024 /srv/node/images/file.img  
meta-data=/srv/node/images/file.img isize=1024 agcount=4, agsize=3276800 blks  
          = sectsz=512 attr=2, projid32bit=0  
data     = bsize=4096 blocks=13107200, imaxpct=25  
          = sunit=0 swidth=0 blks  
naming   =version 2 bsize=4096 ascii-ci=0  
log      =internal log bsize=4096 blocks=6400, version=2  
          = sectsz=512 sunit=0 blks, lazy-count=1  
realtime =none extsz=4096 blocks=0, rtextents=0
```

file コマンドで確認すると以下のとおりになる。

```
$ file /srv/node/images/file.img  
/srv/node/images/file.img: SGI XFS filesystem data (blksz 4096, inosz 1024, v2 dirs)
```

マウント

マウント先のディレクトリを作成する。

```
$ sudo mkdir -p /srv/node/d1
```

マウントする。

```
$ sudo mount -t xfs -o loop,noatime,nodiratime,nobarrier,logbufs=8 /srv/node/images/file.img  
/srv/node/d1
```



df コマンドで確認する。

```
$ df -h | grep loop0
/dev/loop0      50G   33M   50G   1% /srv/node/d1
```

これで、ループバックデバイスのマウントが完了する。

/etc/rsyncd.conf を編集して以下を追記する。

```
uid = swift
gid = swift
log file = /var/log/rsyncd.log
pid file = /var/run/rsyncd.pid
address = MANAGEMENT_INTERFACE_IP_ADDRESS

[account]
max connections = 2
path = /srv/node/
read only = false
lock file = /var/lock/account.lock

[container]
max connections = 2
path = /srv/node/
read only = false
lock file = /var/lock/container.lock

[object]
max connections = 2
path = /srv/node/
read only = false
lock file = /var/lock/object.lock
```

MANAGEMENT_INTERFACE_IP_ADDRESS には、各ホストの IP アドレスを入れる。また、max connections の値は、増やすことをおすすめする。

/etc/default/rsync を開いて RSYNC_ENABLE を true にする。

```
RSYNC_ENABLE=true
```

rsync デーモンを起動する。

```
# service rsync start
```

account-server、container-server の設定

account-server、container-server の稼働するホストで各々必要なパッケージをインストールする。注意する点は、memcached をインストールして起動しておく必要がある点である。

```
# apt-get install swift swift-account swift-container memcached
```

/etc/swift に移動し、account-server.conf-sample、container-server.conf-sample、container-reconciler.conf-sample を GitHub から取得してくる。object-server 関連の conf を取得する必要はない。むしろ、object-server 関連の conf が存在すると、最後にプロセス起動を swift-init を用いて行う際に、object-server のプロセスも稼働してしまう。

```
# curl -o /etc/swift/account-server.conf
https://git.openstack.org/cgit/openstack/swift/plain/etc/account-server.conf-
sample?h=stable/liberty
# curl -o /etc/swift/container-server.conf
https://git.openstack.org/cgit/openstack/swift/plain/etc/container-server.conf-
sample?h=stable/liberty
```

/etc/swift/account-server.conf を以下の通り編集する。

```
[DEFAULT]
...
bind_ip = MANAGEMENT_INTERFACE_IP_ADDRESS
bind_port = 6002
user = swift
swift_dir = /etc/swift
devices = /srv/node
mount_check = true
...
[pipeline:main]
pipeline = healthcheck recon account-server
```



```
...
[filter:recon]
use = egg:swift#recon
...
recon_cache_path = /var/cache/swift
/etc/swift/container-server.conf を以下の通り編集する。
[DEFAULT]
...
bind_ip = MANAGEMENT_INTERFACE_IP_ADDRESS
bind_port = 6001
user = swift
swift_dir = /etc/swift
devices = /srv/node
mount_check = true
...
[pipeline:main]
pipeline = healthcheck recon container-server
...
[filter:recon]
use = egg:swift#recon
...
recon_cache_path = /var/cache/swift
account-server、container-server の実データがストアされるディレクトリの owner を変更する。
# chown -R swift:swift /srv/node
recon 用ディレクトリを作成し owner を変更する。
# mkdir -p /var/cache/swift
# chown -R root:swift /var/cache/swift
最後に、memcached は再起動しておこう。
# service memcached restart
```

object-server の設定

object-server の稼働するホストで各々必要なパッケージをインストールする。

```
# apt-get install swift swift-account swift-object
```

/etc/swift に移動し、object-server.conf-sample、object-expirer.conf-sample を GitHub から取得してくる。こちらでは、逆に、account-server、container-server 関連の conf を取得する必要はない。

```
# curl -o /etc/swift/object-server.conf
https://git.openstack.org/cgit/openstack/swift/plain/etc/object-server.conf-
sample?h=stable/liberty
```

/etc/swift/object-server.conf を以下の通り編集する。

```
[DEFAULT]
...
bind_ip = MANAGEMENT_INTERFACE_IP_ADDRESS
bind_port = 6000
user = swift
swift_dir = /etc/swift
devices = /srv/node
mount_check = true
...
[pipeline:main]
pipeline = healthcheck recon object-server
...
[filter:recon]
use = egg:swift#recon
...
recon_cache_path = /var/cache/swift
recon_lock_path = /var/lock
```

object-server の実データがストアされるディレクトリの owner を変更する。

```
# chown -R swift:swift /srv/node
```

recon 用ディレクトリを作成し owner を変更する。

```
# mkdir -p /var/cache/swift
# chown -R swift:swift /var/cache/swift
```



ring の設定

proxy-server のホスト上で ring ファイルを生成する。このタイミングで、クラスターのパーティション数や冗長度が決定される。クラスターの規模によりパーティション数や冗長度は慎重に決める必要がある。パーティション数に関する知見は GREE さんのこの記事がとても役に立つ。[OpenStack Swift による画像ストレージの運用⁸⁾]

今回は、冗長度 3、パーティション数は 2^{17} とする。proxy-server で以下を実行する。コマンドオプションについては、公式ドキュメントを参照する。[Create initial rings⁹⁾]

```
# cd /etc/swift
# rm -f *.ring.gz backups/*.builder backups/*.ring.gz
# swift-ring-builder account.builder create 17 3 1

# swift-ring-builder account.builder add --region 1 --zone 1 --ip 192.168.0.3 --port 6002 --
device d1 --weight 100
# swift-ring-builder account.builder add --region 1 --zone 2 --ip 192.168.0.4 --port 6002 --
device d1 --weight 100
# swift-ring-builder account.builder add --region 1 --zone 3 --ip 192.168.0.5 --port 6002 --
device d1 --weight 100
# swift-ring-builder account.builder add --region 1 --zone 4 --ip 192.168.0.6 --port 6002 --
device d1 --weight 100
# swift-ring-builder account.builder add --region 1 --zone 5 --ip 192.168.0.7 --port 6002 --
device d1 --weight 100

# swift-ring-builder account.builder
# swift-ring-builder account.builder rebalance

# swift-ring-builder container.builder create 17 3 1

# swift-ring-builder container.builder add --region 1 --zone 1 --ip 192.168.0.3 --port 6001 --
device d1 --weight 100
# swift-ring-builder container.builder add --region 1 --zone 2 --ip 192.168.0.4 --port 6001 --
device d1 --weight 100
# swift-ring-builder container.builder add --region 1 --zone 3 --ip 192.168.0.5 --port 6001 --
device d1 --weight 100
# swift-ring-builder container.builder add --region 1 --zone 4 --ip 192.168.0.6 --port 6001 --
device d1 --weight 100
# swift-ring-builder container.builder add --region 1 --zone 5 --ip 192.168.0.7 --port 6001 --
device d1 --weight 100

# swift-ring-builder container.builder
# swift-ring-builder container.builder rebalance

# swift-ring-builder object.builder create 17 3 1

# swift-ring-builder object.builder add --region 1 --zone 1 --ip 192.168.0.8 --port 6000 --
device d1 --weight 100
# swift-ring-builder object.builder add --region 1 --zone 2 --ip 192.168.0.9 --port 6000 --
device d1 --weight 100
# swift-ring-builder object.builder add --region 1 --zone 3 --ip 192.168.0.10 --port 6000 --
device d1 --weight 100
# swift-ring-builder object.builder add --region 1 --zone 4 --ip 192.168.0.11 --port 6000 --
device d1 --weight 100
# swift-ring-builder object.builder add --region 1 --zone 5 --ip 192.168.0.12 --port 6000 --
device d1 --weight 100

# swift-ring-builder object.builder
# swift-ring-builder object.builder rebalance
```

完了したら、account.ring.gz、container.ring.gz、object.ring.gz を各ノードに配置する。

プロセスの起動

GitHub から swift.conf を取得して全ノードに配る。

```
# curl -o /etc/swift/swift.conf \
https://git.openstack.org/cgit/openstack/swift/plain/etc/swift.conf-sample?h=stable/liberty
/etc/swift/swift.conf を編集する。suffix/prefix の文字列は任意のもので良いが決して外部に漏らしてはならない。
```

⁸⁾<http://labs.gree.jp/blog/2014/12/11746/>

⁹⁾<http://docs.openstack.org/kilo/install-guide/install/apt/content/swift-initial-rings.html>



[swift-hash]

...

swift_hash_path_suffix = HASH_PATH_PREFIX

swift_hash_path_prefix = HASH_PATH_SUFFIX

...

[storage-policy:0]

...

name = Policy-0

default = yes

編集が終わったタイミングで、swift.conf を全ノードにコピーする。

その後、全ノードで /etc/swift の owner を変更する。

```
# chown -R root:swift /etc/swift
```

proxy-server のホストでは、proxy-server と memcached が起動していれば良い。それ以外のホストでは、swift-init がよしなにやってくれるはずである。

```
# service memcached restart  
# service swift-proxy restart
```

その他のサーバでは、以下のコマンドを実行する。

```
# swift-init all start
```

構築できているかどうかの確認

Kilo リリースであるため Keystone の API は v3 を利用している。そのため、swift コマンドを実行する際には、-V3 というオプションが必要になる。

```
$ source demo-openrc.sh  
$ swift stat  
    Account: AUTH_25e9c03ea9824a6e8d24a60ac5e72c98  
Containers: 0  
    Objects: 0  
    Bytes: 0  
Containers in policy "policy-0": 0  
    Objects in policy "policy-0": 0  
    Bytes in policy "policy-0": 0  
X-Account-Project-Domain-Id: default  
    Connection: keep-alive  
X-Timestamp: 1441783575.55310  
X-Trans-Id: tx2260ad3b3ed840f99d075-0056091154  
Content-Type: text/plain; charset=utf-8  
Accept-Ranges: bytes
```

このようにプロンプトが返ってくれば問題ない。プロンプトが少し時間がたっても返ってこない場合、どこかで設定が間違っているはずである。その際には、まず --debug オプションを付けて再度実行してみてほしい。

```
$ swift --debug stat
```

内部で実行されている REST API の詳細が分かる。あとは、各ホストのログを確認していくべきだ。

以上で、構築は完了である。



Chapter 5. Swift の運用と管理

現在の Ring 情報の確認

```
# swift-ring-builder /etc/swift/account.builder
/etc/swift/account.builder, build version 5
131072 partitions, 3.000000 replicas, 1 regions, 5 zones, 5 devices, 0.00 balance, 0.00
dispersion
The minimum number of hours before a partition can be reassigned is 1
The overload factor is 0.00% (0.000000)
Devices: id region zone ip address port replication ip replication port name
weight partitions balance meta
      0      1      1    192.168.0.3   6002      192.168.0.3          6002    d1
100.00  78643 -0.00
      1      1      2    192.168.0.4   6002      192.168.0.4          6002    d1
100.00  78643 -0.00
      2      1      3    192.168.0.5   6002      192.168.0.5          6002    d1
100.00  78644  0.00
      3      1      4    192.168.0.6   6002      192.168.0.6          6002    d1
100.00  78643 -0.00
      4      1      5    192.168.0.7   6002      192.168.0.7          6002
d1 100.00  78643 -0.00
# swift-ring-builder /etc/swift/container.builder
/etc/swift/container.builder, build version 5
131072 partitions, 3.000000 replicas, 1 regions, 5 zones, 5 devices, 0.00 balance, 0.00
dispersion
The minimum number of hours before a partition can be reassigned is 1
The overload factor is 0.00% (0.000000)
Devices: id region zone ip address port replication ip replication port name
weight partitions balance meta
      0      1      1    192.168.0.3   6001      192.168.0.3          6001    d1
100.00  78643 -0.00
      1      1      2    192.168.0.4   6001      192.168.0.4          6001    d1
100.00  78643 -0.00
      2      1      3    192.168.0.5   6001      192.168.0.5          6001    d1
100.00  78644  0.00
      3      1      4    192.168.0.6   6001      192.168.0.6          6001    d1
100.00  78643 -0.00
      4      1      5    192.168.0.7   6001      192.168.0.7          6001
d1 100.00  78643 -0.00
# swift-ring-builder /etc/swift/object.builder
/etc/swift/object.builder, build version 5
131072 partitions, 3.000000 replicas, 1 regions, 5 zones, 5 devices, 0.00 balance, 0.00
dispersion
The minimum number of hours before a partition can be reassigned is 1
The overload factor is 0.00% (0.000000)
Devices: id region zone ip address port replication ip replication port name
weight partitions balance meta
      0      1      1    192.168.0.8   6000      192.168.0.8          6000    d1
100.00  78643 -0.00
      1      1      2    192.168.0.9   6000      192.168.0.9          6000    d1
100.00  78643 -0.00
      2      1      3    192.168.0.10  6000      192.168.0.10          6000    d1
100.00  78643 -0.00
      3      1      4    192.168.0.11  6000      192.168.0.11          6000    d1
100.00  78643 -0.00
      4      1      5    192.168.0.12  6000      192.168.0.12          6000    d1
100.00  78644  0.00
```

デバイス削除とデバイス追加

リバランス方法



クラスター構築時に Ring ファイルを生成した proxy-server 上で rebalance コマンドを実行する。すると、新しく Ring ファイルが作成されるのでそれを再度全ノードに配布する。これは、object のリバランスの例である。

デバイス削除

id4 のデバイスを以下のコマンドで削除する。id4 の場合、対象は d4 である。

```
# swift-ring-builder /etc/swift/object.builder remove d4
d4r1z5-192.168.0.12:6000R192.168.0.12:6000/d1_"" marked for removal and will be removed next
rebalance.
```

リバランス

```
# swift-ring-builder /etc/swift/object.builder rebalance
Reassigned 78644 (60.00%) partitions. Balance is now 0.00. Dispersion is now 0.00
```

Ring 確認

```
# swift-ring-builder /etc/swift/object.builder
/etc/swift/object.builder, build version 5
131072 partitions, 3.000000 replicas, 1 regions, 4 zones, 4 devices, 0.00 balance, 0.00
dispersion
The minimum number of hours before a partition can be reassigned is 1
The overload factor is 0.00% (0.000000)
Devices: id region zone      ip address   port  replication ip  replication port    name
weight partitions balance meta
          0       1     1      192.168.0.8  6000    192.168.0.8        6000    d1
100.00    78643 -0.00
          1       1     2      192.168.0.9  6000    192.168.0.9        6000    d1
100.00    78643 -0.00
          2       1     3      192.168.0.10 6000    192.168.0.10        6000    d1
100.00    78643 -0.00
          3       1     4      192.168.0.11 6000    192.168.0.11        6000    d1
100.00    78643 -0.00
```

その後、生成された object.ring.gz を scp なりして、全ノードに配布する。配布された瞬間からリバランスが各ノードで開始される。これで削除されたデバイス分のデータが残りのノードで再分配される。

デバイス追加

初期構築時と同じコマンドで良い。

```
# swift-ring-builder object.builder add --region 1 --zone 5 --ip 192.168.0.12 --port 6000 --
device d1 --weight 100
```

リバランス

```
# swift-ring-builder object.builder rebalance
Reassigned 78640 (60.00%) partitions. Balance is now 0.00. Dispersion is now 0.00
```

Ring 確認

```
# swift-ring-builder /etc/swift/object.builder
/etc/swift/object.builder, build version 5
131072 partitions, 3.000000 replicas, 1 regions, 5 zones, 5 devices, 0.00 balance, 0.00
dispersion
The minimum number of hours before a partition can be reassigned is 1
The overload factor is 0.00% (0.000000)
Devices: id region zone      ip address   port  replication ip  replication port    name
weight partitions balance meta
          0       1     1      192.168.0.8  6000    192.168.0.8        6000    d1
100.00    78643 -0.00
          1       1     2      192.168.0.9  6000    192.168.0.9        6000    d1
100.00    78643 -0.00
          2       1     3      192.168.0.10 6000    192.168.0.10        6000    d1
100.00    78643 -0.00
          3       1     4      192.168.0.11 6000    192.168.0.11        6000    d1
100.00    78643 -0.00
          4       1     5      192.168.0.12 6000    192.168.0.12        6000    d1
100.00    78644  0.00
```

運用で必ず必要になる基本的な操作は以上である。



OpenStack の本 中級編

「ではみなさん、そういうふうに川だと云われたり、乳の流れたあとだと云われたりしていたこのほんやりと白いものがほんとうは何か承知ですか。」先生は、黒板に吊した大きな黒い星座の図の、上から下へ白くけぶった銀河帯のようなところを指しながら、みんなに問をかけました。

ところが先生は早くもそれを見附けたのでした。

やっぱり星だとジョバンニは思いましたがこんどもすぐに答えることができませんでした。

やっぱり星だとジョバンニは思いましたがこんどもすぐに答えることができませんでした。

ジョバンニは勢よく立ちあがりましたが、立って見るともうはっきりとそれを答えることができないのでした。ザネリが前の席からふりかえって、ジョバンニを見てくすっとわらいました。ジョバンニはもうどぎまぎしてまっ赤になってしまいました。先生がまた云いました。

先生は意外なようにしばらくじっとカムバネルラを見ていましたが、急いで「では。よし」と云いながら、自分で星図を指しました。

ジョバンニは勢よく立ちあがりましたが、立って見るともうはっきりとそれを答えることができないのでした。ザネリが前の席からふりかえって、ジョバンニを見てくすっとわらいました。ジョバンニはもうどぎまぎしてまっ赤になってしまいました。先生がまた云いました。

「このほんやりと白い銀河を大きいい望遠鏡で見ますと、もうたくさんの小さな星に見えるのです。ジョバンニさんそうでしょう。」

カムバネルラが手をあげました。それから四五人手をあげました。ジョバンニも手をあげようとして、急いでそのままやめました。たしかにあれがみんな星だと、いつか雑誌で読んだのですが、このごろはジョバンニはまるで毎日教室でもねむく、本を読むひまも読む本もないでの、なんだかどんなこともよくわからないという気持ちがするのでした。

先生はしばらく困ったようでしたが、眼をカムバネルラの方へ向けて、

制作：Project-VI

著者：こじろう・まっきー

挿絵：かとう

発行：2015年12月31日

印刷：POPLS (<http://www.inv.co.jp/popls/>)



